БИБЛИОТЕКА ПРОГРАММИСТА

Йон Снейдер

Эффективное программирование TCP/IP

- Особенности протоколов TCP/IP
- Идеи и концепции
- Полезные советы
- Технические приемы
- Эвристические правила программирования
- Многочисленные примеры







Йон Снейдер

БИБЛИОТЕКА ПРОГРАММИСТА

Эффективное программирование TCP/IP

Программирование TCP/IP может показаться очень простым, но это заблуждение. Многие программисты сталкиваются с тем, что написанное ими сетевое приложение недостаточно надежно. Часто причиной такого положения дел является неосторожное обращение с сетевыми протоколами. Поэтому основное внимание в данной книге уделено тонким вопросам функционирования семейства протоколов и способам работы с ними. Здесь изложены подтвержденные практикой советы, технические приемы и эвристические правила программирования TCP/IP для достижения максимальной производительности; показано, как избежать многих типичных ошибок. Каркас кода и библиотека функций дают возможность создавать приложения, не думая о низкоуровневых деталях. Основные идеи и концепции иллюстрируются многочисленными примерами. Вы станете лучше понимать внутреннее устройство TCP/IP и получите необходимые практические навыки работы с этим семейством протоколов.

- Терминология
- Особенности протоколов TCP/IP
- Идеи и концепции
- Полезные советы
- Технические приемы
- Эвристические прввила progrвммирования
- Создание эффективных и надежных программ
- Виды интерфейса
- Инструменты и ресурсы
- Многочисленные примеры

Уровень пользователя:
опытный/эксперт
Серия:
библиотека программиста









Jon C. Snader

Effective TCP/IP Programming

44 Tips to Improve Your Network Programs



Addison-Wesley
An imprint of Addison Wesley Longman, Inc.
Reading, Masachusetts • Harlow, England • Menlo Park, California
Berkley, California • Don Mills, Ontario • Sydney
Bonn • Amsterdam • Tokyo • Mexico City

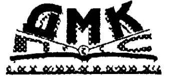
Йон Снейдер

БИБЛИОТЕКА ПРОГРАММИСТА

Эффективное программирование TCP/IP



Санкт-Петербург
Москва • Харьков • Минск
2002



Й. Снейдер

Эффективное программирование TCP/IP
Библиотека программиста

Перевел с английского А. Слинкин

Главный редактор	*Е. Строганова*
Заведующий редакцией	*И. Корнеев*
Руководитель проекта	*И. Захаров*
Научный редактор	*М. Нилов*
Литературный редактор	*Н. Морозова*
Технический редактор	*А. Кукушкина*
Иллюстрации	*А. Бахарев*
Художник	*Н. Биржаков*
Верстка	*А. Татаринов*

ББК 32.973.202-018.2

Снейдер Й.

С53 Эффективное программирование TCP/IP. Библиотека программиста. — СПб.: Питер, 2002. — 320 с.: ил.

ISBN 5-318-00453-9

Программирование TCP/IP может показаться очень простым, но это заблуждение. Многие программисты сталкиваются с тем, что написанное ими сетевое приложение недостаточно надежно. Часто причиной такого положения дел является неосторожное обращение с сетевыми протоколами. Поэтому основное внимание в данной книге уделено тонким вопросам функционирования семейства протоколов и способам работы с нним. Здесь изложены подтвержденные практикой советы, технические приемы и эвристические правила программирования TCP/IP для достижения максимальной производительности; показано, как избежать многих типичных ошибок. Каркас кода и библиотека функций дают возможность создавать приложения, не думая о низкоуровневых деталях. Основные идеи и концепции иллюстрируются многочисленными примерами. Вы станете лучше понимать внутреннее устройство TCP/IP и получите необходимые практические навыки работы с этим семейством протоколов.

Оригинал-макет подготовлен издательством «ДМК Пресс».

ISBN 5-318-00453-9

ISBN 0-201-61589-4 (англ.)

ЗАО «Питер Бук». 196105, Санкт-Петербург, Благодатная ул., д. 67.
Лицензия ИД № 01940 от 05.06.00.

Налоговая льгота – общероссийский классификатор продукции ОК 005-93, том 2; 953000 – книги и брошюры.

Подписано в печать 18.12.01. Формат 70×100 1/16. Усл. п. л. 25,8. Доп. тираж 5000 экз. Заказ № 2452.

Отпечатано с фотоформ в ФГУП «Печатный двор» им. А. М. Горького Министерства РФ
по делам печати, телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.

Содержание

Посвящается Марии

Предисловие

В результате взрывного развития Internet, беспроводных видов связи и сетей соразмерно увеличилось число программистов и инженеров, занимающихся разработкой сетевых приложений. Программирование TCP/IP может показаться обманчиво простым. Интерфейс прикладного программирования (API) несложен. Даже новичок может взять шаблон клиента или сервера и создать на его основе работающее приложение.

К сожалению, нередко после весьма продуктивного начала неофиты начинают понимать, что все не так очевидно, а созданная ими программа оказывается и медленной, и нестабильной. В сетевом программировании есть множество «темных уголков» и трудно понимаемых деталей. Цель этой книги – ответить на возникающие вопросы и помочь разобраться с тонкостями программирования TCP/IP.

Прочитав данную книгу, вы научитесь преодолевать трудности сетевого программирования. Здесь будут рассмотрены многие вопросы, на первый взгляд, лишь отдаленно связанные с теми знаниями, которыми должен обладать программист сетевых приложений. Но без понимания таких деталей не разобраться в том, как сетевые протоколы взаимодействуют с приложением. Ранее казавшееся загадочным «поведение» приложения при ближайшем рассмотрении становится совершенно понятным, решение проблемы лежит на поверхности.

Книга построена несколько необычно. Типичные проблемы представлены в виде серии советов. Разбираясь с конкретным вопросом, вы будете переходить к изучению того или иного аспекта TCP/IP. К концу главы вы не только решите частную задачу, но и углубите понимание того, как работают и взаимодействуют с приложением протоколы TCP/IP.

Разбивка книги на отдельные советы в какой-то мере лишает текст логической последовательности. Чтобы помочь вам сориентироваться, в главу 1 помещен путеводитель, описывающий расположение материала. Получить общее представление об организации книги поможет также оглавление, в котором перечислены все советы. Поскольку каждый совет дан в повелительном наклонении, оглавление можно считать списком рецептов.

С другой стороны, такая организация материала позволяет использовать книгу в качестве справочника. Столкнувшись в повседневной работе с какой-либо проблемой, вы можете обратиться к соответствующему совету, чтобы вспомнить

ее решение. Многие темы затрагиваются в нескольких советах, иногда под разным углом зрения. Такое повторение помогает усвоению сложных концепций, проясняя мотивы тех или иных решений.

Аудитория

Данная книга предназначена, главным образом, начинающим и программистам среднего уровня, но даже опытные специалисты найдут в ней много полезного для себя. Хотя и предполагается, что читатель знаком с сетями и основами API на базе сокетов, в главе 1 приводится обзор элементарных вызовов API и их использования для создания примитивного клиента и сервера. В совете 4 более детально рассмотрены модели клиента и сервера, поэтому даже читатель с минимальной подготовкой сможет извлечь из представленного материала практическую пользу.

Почти все примеры написаны на языке C, безусловно, необходимы базовые навыки программирования на этом языке для понимания приведенных в книге программ. В совете 31 представлены некоторые примеры на языке Perl. Но, впрочем, предварительное знание Perl необязательно. Здесь встречаются и небольшие примеры на языках командных интерпретаторов (shell), но и для их понимания знакомства с shell-программированием не нужно.

Материал для изучения подан по возможности максимально независимо от платформы. За немногими исключениями, приводимые в примерах программы должны компилироваться и работать на любой платформе UNIX или Win32. Но программисты, которые используют системы, отличные от UNIX и Windows, тоже могут без особых трудностей применять примеры на своей платформе.

Принятые в книге соглашения

По ходу изложения приведены небольшие программы для иллюстрации различных аспектов исследуемой проблемы. Здесь будут использоваться следующие соглашения:

- текст, который набирает пользователь, печатается **`полужирным моноширинным шрифтом`**;
- текст, который выводят системы, печатается `обычным моноширинным шрифтом`;
- комментарии, не являющиеся частью ввода или вывода, печатаются *`курсивным моноширинным шрифтом`*.

Пример из совета 9:

```
bsd: $ tcprw localhost 9000
hello
получено сообщение 1    печатается после пятисекундной задержки
                        здесь сервер остановили
hello again
tcprw: ошибка вызова readline: Connection reset by peer (54)
bsd: $
```

Обратите внимание на приглашение командного интерпретатора, содержащее имя системы. Предыдущий пример исполнялся на машине с именем bsd.

В рамке дается описание вводимой в рассмотрение новой функции API – собственной или системного вызова. Стандартные системные вызовы обводятся сплошной рамкой:

```
#include <sys/socket.h>   /* UNIX */
#include <winsock2.h>     /* Windows */

int connect( SOCKET s, const struct sockaddr *peer, int peer_len );
```

Возвращаемое значение: 0 – нормально, –1 (UNIX) или не 0 (Windows) – ошибка.

Разработанные автором функции обведены пунктирной рамкой:

```
#include "etcp.h"

SOCKET tcp_server( char *host, char *port );
```

Возвращаемое значение: сокет в режиме прослушивания (в случае ошибки завершает программу).

Примечание *Таким образом отмечены попутные замечания и дополнительный материал. При первом чтении подобный материал можно пропускать.*

Наконец, URL подчеркивается:
http://www.freebsd.org.

Исходные тексты и список исправлений

Исходные тексты всех встречающихся в книге примеров представлены на сайте издательства «ДМК-Пресс» http://www.dmk.ru. Вы можете загрузить их на свой компьютер и поэкспериментировать. На этом сайте находятся каркасы программ и код библиотечных функций.

Оформление

Мне очень нравится оформление книг Ричарда Стивенса*. Приступая к работе над этой книгой, я попросил у Рича разрешения скопировать его стиль. Рич со свойственным ему великодушием не только не возражал, но даже посодействовал этой «краже», прислав мне макросы для форматера GROFF, которыми он пользовался при наборе своих книг.

Если вам понравится это издание, то благодарить следует Рича. В противном случае загляните в любую из его книг, чтобы понять, к чему я стремился.

* Речь идет о трехтомном издании «TCP\IP Illustrated» и двухтомном «UNIX Network Programming». – *Прим. перев.*

Благодарности

Традиционно авторы книг благодарят свои семьи за поддержку во время работы над книгой, и теперь я знаю, почему. Эта работа не была бы закончена, если бы не помощь моей жены Марии. И эти слова – лишь жалкая попытка воздать ей долж-ное за дополнительные хлопоты и одинокие вечера.

Также ценнейшую поддержку оказали рецензенты. В ранних вариантах рукописи они нашли многочисленные ошибки, как технические, так и типографские, разъяснили то, что я неправильно понимал или не знал, предложили свежий подход. Хочется выразить благодарность Крису Клиланду (Chris Cleeland), Бобу Джиллигену (Bob Gilligan, FreeGate Corp.), Питеру Хэверлоку (Peter Haverlock, Nortel Networks), С. Ли Генри (S. Lee Henry, Web Publishing, Inc.), Мукешу Кэкеру (Mukesh Kacker, Sun Microsystems, Inc.), Барри Марголину (Barry Margolin, GTE Internetworking), Майку Оливеру (Mike Oliver, Sun Microsystems, Inc.), Юри Рацу (Uri Raz) и Ричу Стивенсу (Rich Stevens).

Необходимо поблагодарить редактора Карен Геттман (Karen Gettman), ведущего редактора Мэри Гарт (Mary Hart), координатора проекта Тиреллу Элбо (Tyrrell Albaugh) и корректора Кэт Охала (Cat Ohala). Мне было приятно с ними работать, они очень помогли начинающему автору.

Я приветствую любые замечания и предложения читателей. Пишите мне по электронному адресу, указанному ниже.

Йон Снейдер

Тампа, Флорида
Декабрь, 1999

jsnader@ix.netcom.com
http://www.netcom.com/~jsnader

Глава 1. Введение

Цель этой книги – помочь программистам разных уровней – от начального до среднего – повысить свою квалификацию. Для получения статуса мастера требуется практический опыт и накопление знаний в конкретной области. Конечно, опыт приходит только со временем и практикой, но данная книга существенно пополнит багаж ваших знаний.

Сетевое программирование – это обширная область с большим выбором различных технологий для желающих установить связь между несколькими машинами. Среди них такие простые, как последовательная линия связи, и такие сложные, как системная сетевая архитектура (SNA) компании IBM. Но сегодня протоколы TCP/IP – наиболее перспективная технология построения сетей. Это обусловлено развитием Internet и самого распространенного приложения – Всемирной паутины (World Wide Web).

Примечание *Вообще-то, Web – не приложение. Но это и не протокол, хотя в ней используются и приложения (Web-браузеры и серверы), и протоколы (например, HTTP). Web – это самое популярное среди пользователей Internet применение сетевых технологий.*

Однако и до появления Web TCP/IP был распространенным методом создания сетей. Это открытый стандарт, и на его основе можно объединять машины разных производителей. К концу 90-х годов TCP/IP завоевал лидирующее положение среди сетевых технологий, видимо, оно сохранится и в дальнейшем. По этой причине в книге рассматриваются TCP/IP и сети, в которых он работает.

При желании совершенствоваться в сетевом программировании необходимо сначала овладеть некоторыми основами, чтобы в полной мере оценить, чем же вам предстоит заниматься. Рассмотрим несколько типичных проблем, с которыми сталкиваются начинающие. Многие из этих проблем – результат частичного или полного непонимания некоторых аспектов протоколов TCP/IP и тех API, с помощью которых программа использует эти протоколы. Такие проблемы возникают в реальной жизни и порождают многочисленные вопросы в сетевых конференциях.

Некоторые термины

За немногими исключениями, весь материал этой книги, в том числе примеры программ, предложен для работы в системах UNIX (32 и 64-разрядных) и системах, использующих API Microsoft Windows (Win32 API). Я не экспериментировал с 16-разрядными приложениями Windows. Но и для других платформ почти все остается применимым.

Желание сохранить переносимость привело к некоторым несообразностям в примерах программ. Так, программисты, работающие на платформе UNIX, неодобрительно отнесутся к тому, что для дескрипторов сокетов применяется тип `SOCKET` вместо привычного `int`. А программисты Windows заметят, что я ограничился только консольными приложениями. Все принятые соглашения описаны в совете 4.

По той же причине я обычно избегаю системных вызовов `read` и `write` для сокетов, так как Win32 API их не поддерживает. Для чтения из сокета или записи в него применяются системные вызовы `recv`, `recvfrom` или `recvmsg` для чтения и `send`, `sendto` или `sendmsg` для записи.

Одним из самых трудных был вопрос о том, следует ли включать в книгу материал по протоколу IPv6, который в скором времени должен заменить современную версию протокола IP (IPv4). В конце концов, было решено не делать этого. Тому есть много причин, в том числе:

- почти все изложенное в книге справедливо как для IPv4, так и для IPv6;
- различия, которые все-таки имеются, по большей части сосредоточены в тех частях API, которые связаны с адресацией;
- книга представляет собой квинтэссенцию опыта и знаний современных сетевых программистов, а реального опыта работы с протоколом IPv6 еще не накоплено.

Поэтому, если речь идет просто об IP, то подразумевается IPv4. Там, где упоминается об IPv6, об этом написано.

И, наконец, я называю восемь бит информации *байтом*. В сетевом сообществе принято называть такую единицу *октетом* – по историческим причинам. Когда-то размер байта зависел от платформы, и не было единого мнения о его точной длине. Чтобы избежать неоднозначности, в ранней литературе по сетям и был придуман термин *октет*. Но сегодня все согласны с тем, что длина байта равна восьми битам [Kernighan and Pike 1999], так что употребление этого термина можно считать излишним педантизмом.

Примечание *Однако утверждения о том, что длина байта равна восьми битам, время от времени все же вызывают споры в конференциях Usenet: «Ох уж эта нынешняя молодежь! Я в свое время работал на машине Баста-6, в которой байт был равен пяти с половиной битам. Так что не рассказывайте мне, что в байте всегда восемь бит».*

Путеводитель по книге

Ниже будут рассмотрены основы API сокетов и архитектура клиент-сервер, свойственная приложениям, в которых используется TCP/IP. Это тот фундамент, на котором вы станете возводить здание своего мастерства.

В главе 2 обсуждаются некоторые заблуждения по поводу TCP/IP и сетей вообще. В частности, вы узнаете, в чем разница между протоколами, требующими логического соединения, и протоколами, не нуждающимися в нем. Здесь будет рассказано об IP-адресации и подсетях (эта концепция часто вызывает недоумение), о бесклассовой междоменной маршрутизации (Classless Interdomain Routing – CIDR)

и преобразовании сетевых адресов (Network Address Translation – NAT). Вы увидите, что TCP в действительности не гарантирует доставку данных. И нужно быть готовым к некорректным действиям как пользователя, так и программы на другом конце соединения. Кроме того, приложения будут по-разному работать в глобальной (WAN) и локальной (LAN) сетях.

Следует напомнить, что TCP – это *потоковый* протокол, и разъяснить его значение для программистов. Вы также узнаете, что TCP автоматически не обнаруживает потерю связи, почему это хорошо и что делать в этой ситуации.

Вам будет понятно, почему API сокетов всегда следует предпочитать API на основе интерфейса транспортного уровня (Transport Layer Interface – TLI) и транспортному интерфейсу X/Open (X/Open Transport Interface – XTI). Кроме того, я объясню, почему не стоит слишком уж серьезно воспринимать модель открытого взаимодействия систем (Open Systems Interconnection – OSI). TCP – очень эффективный протокол с отличной производительностью, так что обычно не нужно дублировать его функциональность с помощью протокола UDP.

В главе 2 разработаны каркасы для нескольких видов приложений TCP/IP и на их основе построена библиотека часто используемых функций. Каркасы и библиотека позволяют писать приложения, не заботясь о преобразовании адресов, управлении соединением и т.п. Если каркас готов, то вряд ли следует срезать себе путь, например, «зашив» в код адреса и номера портов или опустив проверку ошибок.

Каркасы и библиотека используются в книге для построения тестов, небольших примеров и автономных приложений. Часто требуется всего лишь добавить пару строк в один из каркасов, чтобы создать специализированную программу или тестовый пример.

В главе 3 подробно рассмотрены некоторые вопросы, на первый взгляд кажущиеся тривиальными. Например, что делает операция записи в контексте TCP. Вроде бы все очевидно: записывается в сокет n байт, а TCP отсылает их на другой конец соединения. Но вы увидите, что иногда это происходит не так. В протоколе TCP есть сложный свод правил, определяющих, можно ли посылать данные немедленно и, если да, то сколько. Чтобы создавать устойчивые и эффективные программы, необходимо усвоить эти правила и их влияние на приложения.

То же относится к чтению данных и завершению соединения. Вы изучите эти операции и разберетесь, как нужно правильно завершать соединение, чтобы не потерять информацию. Здесь будет рассмотрена и операция установления соединения `connect`: когда при ее выполнении возникает тайм-аут и как она используется в протоколе UDP.

Будет рассказано об имеющимся в системе UNIX суперсервере `inetd`, упрощающим написание сетевых приложений. Вы научитесь пользоваться программой `tcpmux`, которая избавляет от необходимости назначать серверам хорошо известные порты. Узнаете, как работает `tcpmux`, и сможете создать собственную версию для систем, где это средство отсутствует.

Кроме того, здесь подробно обсуждаются такие вопросы, как состояние TIME-WAIT, алгоритм Нейгла, выбор размеров буферов и правильное применение опции `SO_REUSEADDR`. Вы поймете, как сделать свои приложения событийно-управляемыми и создать отдельный таймер для каждого события. Будут описаны некоторые

типичные ошибки, которые допускают даже опытные программисты, и приемы повышения производительности.

Наконец, вы познакомитесь с некоторыми языками сценариев, используемыми при программировании сетей. С их помощью можно легко и быстро создавать полезные сетевые утилиты.

Глава 4 посвящена двум темам. Сначала будет рассмотрено несколько инструментальных средств, необходимых каждому сетевому программисту. Показано, как использовать утилиту `ping` для диагностики простейших неисправностей. Затем рассказывается о сетевых анализаторах пакетов (sniffer) вообще и программе `tcpdump` в частности. В этой главе дано несколько примеров применения `tcpdump` для диагностики сетевых проблем. С помощью программы `traceroute` исследуется маленькая часть Internet.

Утилита `ttcp`, в создании которой принимал участие Майк Муусс (Mike Muuss) – автор программы `ping`, является полезным инструментом для изучения производительности сети и влияния на нее тех или иных параметров TCP. Будут продемонстрированы некоторые методы диагностики. Еще одна бесплатная инструментальная программа `lsof` необходима в ситуации, когда нужно сопоставить сетевые соединения с процессами, которые их открыли. Очень часто `lsof` предоставляет информацию, получение которой иным способом потребовало бы поистине героических усилий.

Много внимания уделено утилите `netstat` и той разнообразной информации, которую можно получить с ее помощью, а также программам трассировки системных вызовов, таким как `ktrace` и `truss`.

Обсуждение инструментов диагностики сетей завершается построением утилиты для перехвата и отображения датаграмм протокола ICMP (протокол контроля сообщений в сети Internet). Она не только вносит полезный вклад в ваш набор инструментальных средств, но и иллюстрирует использование простых сокетов (raw sockets).

Во второй части главы 4 описаны дополнительные ресурсы для пополнения знаний о TCP/IP и сетях. Я познакомлю вас с замечательными книгами Ричарда Стивенса, источниками исходных текстов, и собранием документов RFC (предложений для обсуждения), размещенных на сервере проблемной группы проектирования Internet (Internet Engineering Task Force – IETF) и в конференциях Usenet.

Архитектура клиент-сервер

Хотя постоянно говорится о *клиентах* и *серверах*, не всегда очевидно, какую роль играет конкретная программа. Иногда программы являются равноправными участниками обмена информацией, нельзя однозначно утверждать, что одна программа обслуживает другую. Однако в случае с TCP/IP различие более четкое. Сервер прослушивает порт, чтобы обнаружить входящие TCP-соединения или UDP-датаграммы от одного или нескольких клиентов. С другой стороны, можно сказать, что клиент – это тот, кто начинает диалог первым.

В книге рассмотрены три типичных случая архитектуры клиент-сервер, показанные на рис. 1.1. В первом случае клиент и сервер работают на одной машине

(рис. 1.1а). Это самая простая конфигурация, поскольку нет физической сети. Посылаемые данные, передаются стеку TCP/IP, но не помещаются в выходную очередь сетевого устройства, а закольцовываются системой и возвращаются обратно в стек, но уже в качестве принятых данных.

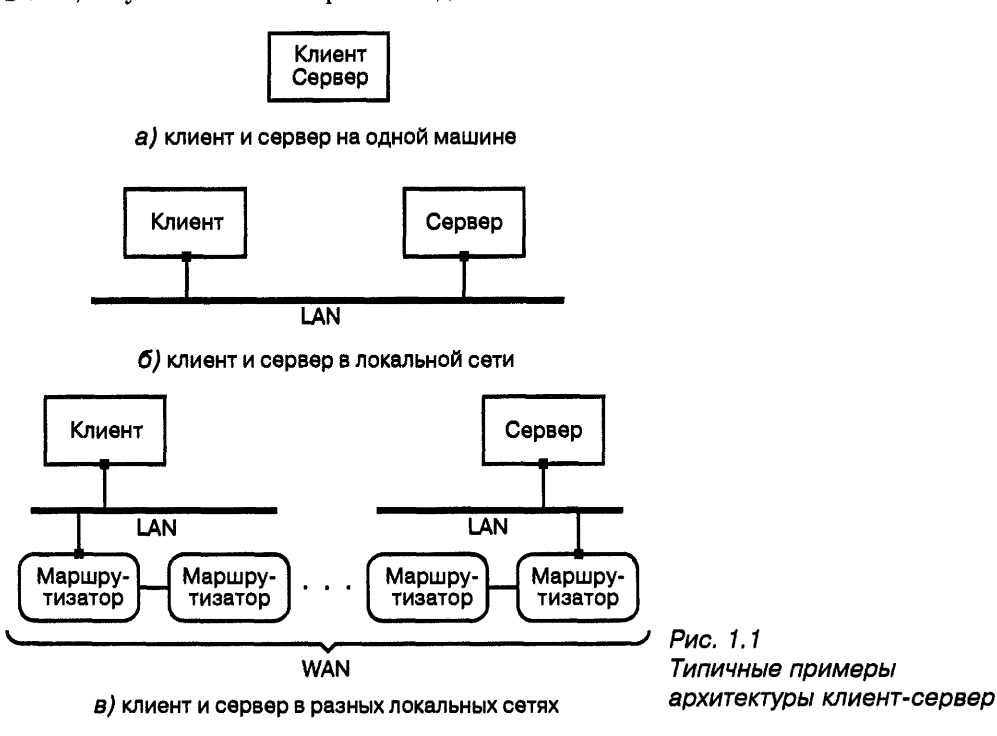


Рис. 1.1 Типичные примеры архитектуры клиент-сервер

На этапе разработки такое размещение клиента и сервера дает определенные преимущества, даже если в реальности они будут работать на разных машинах. Во-первых, проще оценить производительность обеих программ, так как сетевые задержки исключаются. Во-вторых, этот метод создает идеальную лабораторную среду, в которой пакеты не пропадают, не задерживаются и всегда приходят в правильном порядке.

Примечание *По крайней мере, почти всегда. Как вы увидите в совете 7, даже в этой среде можно создать такую нагрузку, что UDP-датаграммы будут пропадать.*

И, наконец, разработку вести проще и удобнее, когда можно все отлаживать на одной машине.

Разумеется, даже в условиях промышленной эксплуатации вполне возможно, что клиент и сервер будут работать на одном компьютере. В совете 26 описана такая ситуация.

Во втором примере конфигурации (рис. 1.1б) клиент и сервер работают на разных машинах, но в пределах одной локальной сети. Здесь имеет место реальная

сеть, но условия все же близки к идеальным. Пакеты редко теряются и практически всегда приходят в правильном порядке. Такая ситуация очень часто встречается на практике. Причем некоторые приложения предназначены для работы только в такой среде.

Типичный пример – сервер печати. В небольшой локальной сети может быть только один такой сервер, обслуживающий несколько машин. Одна машина (или сетевое программное обеспечение на базе TCP/IP, встроенное в принтер) выступает в роли сервера, который принимает запросы на печать от клиентов на других машинах и ставит их в очередь к принтеру.

В третьем примере (рис. 1.1в) клиент и сервер работают на разных компьютерах, связанных глобальной сетью. Этой сетью может быть Internet или корпоративная Intranet, но главное – приложения уже не находятся внутри одной локальной сети, так что на пути IP-датаграмм есть, по крайней мере, один маршрутизатор.

Такое окружение может быть более «враждебным», чем в первых двух случаях. По мере роста трафика в глобальной сети начинают переполняться очереди, в которых маршрутизатор временно хранит поступающие пакеты, пока не отправит их адресату. А когда в очереди больше нет места, маршрутизатор отбрасывает пакеты. В результате клиент должен передавать пакеты повторно, что приводит к появлению дубликатов и доставке пакетов в неправильном порядке. Эти проблемы возникают довольно часто, как вы увидите в совете 38.

О различиях между локальными и глобальными сетями будет рассказано в совете 12.

Элементы API сокетов

В этом разделе кратко рассмотрены основы API сокетов и построены простейшие клиентское и серверное приложения. Хотя эти приложения очень схематичны, на их примере проиллюстрированы важнейшие характеристики клиента и сервера TCP.

Начнем с вызовов API, необходимых для простого клиента. На рис. 1.2 показаны функции, применяемые в любом клиенте. Адрес удаленного хоста задается с помощью структуры `sockaddr_in`, которая передается функции `connect`.

Первое, что вы должны сделать, – это получить сокет для логического соединения. Для этого предназначен системный вызов `socket`.

```
#include <sys/socket.h>  /* UNIX */
#include <winsock2.h>    /* Windows */

SOCKET socket( int domain, int type, int protocol );
```

Возвращаемое значение: дескриптор сокета в случае успеха; –1 (UNIX) или `INVALID_SOCKET` (Windows) – ошибка.

API сокетов не зависит от протокола и может поддерживать разные *адресные домены*. Параметр `domain` – это константа, указывающая, какой домен нужен сокету.

Чаще используются домены AF_INET (то есть Internet) и AF_LOCAL (или AF_UNIX). В книге рассматривается только домен AF_INET. Домен AF_LOCAL применяется для межпроцессного взаимодействия (IPC) на одной и той же машине.

Примечание *Существуют разногласия по поводу того, следует ли обозначать константы доменов* AF_* *или* PF_*. *Сторонники* PF_* *указывают на их происхождение от уже устаревших вариантов вызова* socket *в системах 4.1c/2.8/2.9BSD. И, кроме того, они считают, что PF означает protocol family (семейство протоколов). Сторонники же* AF_* *говорят, что в коде ядра, относящемся к реализации сокетов, параметр* domain *сравнивается именно с константами* AF_*. *Но, поскольку оба набора констант определены одинаково – в действительности одни константы просто выражаются через другие, – на практике можно употреблять оба варианта.*

С помощью параметра *type* задается тип создаваемого сокета. Чаще встречаются следующие значения (а в этой книге только такие) сокетов:

- SOCK_STREAM – обеспечивают надежный дуплексный протокол на основе установления логического соединения. Если говорится о семействе протоколов TCP/IP, то это TCP;
- SOCK_DGRAM – обеспечивают ненадежный сервис доставки датаграмм. В рамках TCP/IP это будет протокол UDP;
- SOCK_RAW –предоставляют доступ к некоторым датаграммам на уровне протокола IP. Они используются в особых случаях, например для просмотра всех ICMP-сообщений.

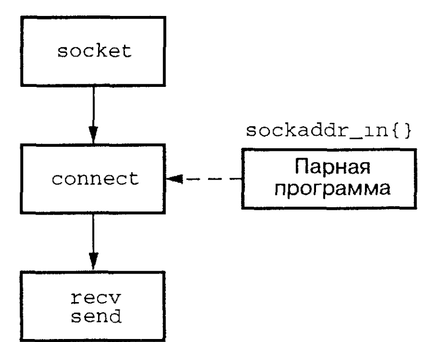


Рис. 1.2. Основные вызовы API сокетов для клиентов

Параметр *protocol* показывает, какой протокол следует использовать с данным сокетом. В контексте TCP/IP он обычно неявно определяется типом сокета, поэтому в качестве значения задают 0. Иногда, например в случае простых (raw) сокетов, имеется несколько возможных протоколов, так что нужный необходимо задавать явно. Об этом будет рассказано в совете 40.

Для самого простого TCP-клиента потребуется еще один вызов API сокетов, обеспечивающий установление соединения:

```
#include <sys/socket.h>  /* UNIX */
#include <winsock2.h>    /* Windows */

int connect( SOCKET s, const struct sockaddr *peer, int peer_len );
```

Возвращаемое значение: 0 – нормально, –1 (UNIX) или не 0 (Windows) – ошибка.

Параметр *s* – это дескриптор сокета, который вернул системный вызов socket. Параметр *peer* указывает на структуру, в которой хранится адрес удаленного хоста и некоторая дополнительная информация. Для домена AF_INET – это структура типа sockaddr_in. Ниже вы увидите, как она заполняется. Параметр *peer_len* содержит размер структуры в байтах, на которую указывает *peer*.

После установления соединения можно передавать данные. В ОС UNIX вы должны обратиться к системным вызовам read и write и передать им дескриптор сокета точно так же, как передали бы дескриптор открытого файла. Увы, как уже говорилось, в Windows эти системные вызовы не поддерживают семантику сокетов, поэтому приходится пользоваться вызовами recv и send. Они отличаются от read и write только наличием дополнительного параметра.

```
#include <sys/socket.h>  /* UNIX */
#include <winsock2.h>    /* Windows */

int recv( SOCKET s, void *buf, size_t len, int flags );

int send( SOCKET s, const void *buf, size_t len, int flags );
```

Возвращаемое значение: число принятых или переданных байтов в случае успеха или –1 в случае ошибки.

Параметры *s*, *buf* и *len* означают то же, что и для вызовов read и write. Значение параметра *flags* в основном зависит от системы, но и UNIX, и Windows поддерживают следующие флаги:

- MSG_OOB – следует послать или принять срочные данные;
- MSG_PEEK – используется для просмотра поступивших данных без их удаления из приемного буфера. После возврата из системного вызова данные еще могут быть получены при последующем вызове read или recv;
- MSG_DONTROUTE – сообщает ядру, что не надо выполнять обычный алгоритм маршрутизации. Как правило, используется программами маршрутизации или для диагностических целей.

При работе с протоколом TCP вам ничего больше не понадобится. Но при работе с UDP нужны еще системные вызовы recvfrom и sendto. Они очень похожи на recv и send, но позволяют при отправке датаграммы задать адрес назначения, а при приеме – получить адрес источника.

```
#include <sys/socket.h>  /* UNIX */
#include <winsock2.h>    /* Windows */

int recvfrom( SOCKET s, void *buf, size_t len, int flags,
      struct sockaddr *from, int *fromlen );

int sendto( SOCKET s, const void *buf, size_t len, int flags,
      const struct sockaddr *to, int tolen);
```

Возвращаемое значение: число принятых или переданных байтов в случае успеха или –1 при ошибке.

Первые четыре параметра – *s*, *buf*, *len* и *flags* – такие же, как в вызовах recv и send. Параметр *from* в вызове recvfrom указывает на структуру, в которую ядро помещает адрес источника пришедшей датаграммы. Длина этого адреса хранится в целом числе, на которое указывает параметр *fromlen*. Обратите внимание, что *fromlen* – это *указатель* на целое.

Аналогично параметр *to* в вызове sendto указывает на адрес структуры, содержащей адреса назначения датаграммы, а параметр *tolen* – длина этого адреса. Заметьте, что *to* – это целое, а не указатель.

В листинге 1.1 приведен пример простого TCP-клиента.

Листинг 1.1. Простейший TCP-клиент

simplec.c

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>

int main( void )
{
    struct sockaddr_in peer;
    int s;
    int rc;
    char buf[ 1 ];

    peer.sin_family = AF_INET;
    peer.sin_port = htons( 7500 );
    peer.sin_addr.s_addr = inet_addr( "127.0.0.1" );

    s = socket( AF_INET, SOCK_STREAM, 0 );
    if ( s < 0 )
    {
       perror( "ошибка вызова socket" );
       exit( 1 );
    }

    rc = connect( s, ( struct sockaddr * )&peer, sizeof( peer ) );
    if ( rc )
    {
       perror( "ошибка вызова connect" );
       exit( 1 );
    }

    rc = send( s, "1", 1, 0 );
    if ( rc <= 0 )
    {
       perror( "ошибка вызова send" );
       exit( 1 );
    }

    rc = recv( s, buf, 1, 0 );
    if ( rc <= 0 )
```

```
35         perror( "ошибка вызова recv" );
36     else
37         printf( "%c\n", buf[ 0 ] );
38     exit( 0 );
39 }
```

simplec.c

Клиент в листинге 1.1 написан как UNIX-программа, чтобы не было сложностей, связанных с переносимостью и Windows-функцией `WSAStartup`. В совете 4 сказано, что в основном эти сложности можно скрыть в заголовочном файле, но сначала надо подготовить некоторые механизмы. Пока ограничимся более простой моделью UNIX.

Подготовка адреса сервера

12-14 Заполняем структуру `sockaddr_in`, записывая в ее поля номер порта (7500) и адрес. 127.0.0.1 – это возвратный адрес, который означает, что сервер находится на той же машине, что и клиент.

Получение сокета и соединение с сервером

15-20 Получаем сокет типа `SOCK_STREAM`. Как было отмечено выше, протокол TCP, будучи потоковым, требует именно такого сокета.

21-26 Устанавливаем соединение с сервером, обращаясь к системному вызову `connect`. Этот вызов нужен, чтобы сообщить ядру адрес сервера.

Отправка и получение одного байта

27-38 Сначала посылаем один байт серверу, затем читаем из сокета один байт и записываем полученный байт в стандартный вывод и завершаем сеанс.

Прежде чем тестировать клиента, необходим сервер. Вызовы API сокетов для сервера немного иные, чем для клиента. Они показаны на рис. 1.3.

Сервер должен быть готов к установлению соединений с клиентами. Для этого он обязан прослушивать известный ему порт с помощью системного вызова `listen`. Но предварительно необходимо привязать адрес интерфейса и номер порта к прослушивающему сокету. Для этого предназначен вызов `bind`:

```
#include <sys/socket.h>  /* UNIX */
#include <winsock2.h>    /* Windows */

int bind( SOCKET s, const struct sockaddr *name, int namelen );
```

Возвращаемое значение: 0 – нормально, –1 (UNIX) или `SOCKET_ERROR` (Windows) – ошибка.

Параметр *s* – это дескриптор прослушивающего сокета. С помощью параметров *name* и *namelen* передаются порт и сетевой интерфейс, которые нужно прослушивать. Обычно в качестве адреса задается константа `INADDR_ANY`. Это означает, что будет принято соединение, запрашиваемое по любому интерфейсу. Если хосту с несколькими сетевыми адресами нужно принимать соединения только по

одному интерфейсу, то следует указать IP-адрес этого интерфейса. Как обычно, *namelen* – длина структуры sockaddr_in.

После привязки локального адреса к сокету нужно перевести сокет в режим прослушивания входящих соединений с помощью системного вызова listen, назначение которого часто не понимают. Его единственная задача – пометить сокет как прослушивающий. Когда хосту поступает запрос на установление соединения, ядро ищет в списке прослушивающих сокетов тот, для которого адрес назначения и номер порта соответствуют указанным в запросе.

```
#include <sys/socket.h>  /* UNIX */
#include <winsock2.h>    /* Windows */

int listen( SOCKET s, int backlog );
```

Возвращаемое значение: 0 – нормально, –1 (UNIX) или SOCKET_ERROR (Windows) – ошибка.

Параметр *s* – это дескриптор сокета, который нужно перевести в режим прослушивания. Параметр *backlog* – это максимальное число ожидающих, но еще не принятых соединений. Следует отметить, что это *не* максимальное число одновременных соединений с данным портом, а лишь максимальное число частично установленных соединений, ожидающих в очереди, пока приложение их примет (описание системного вызова accept дано ниже).

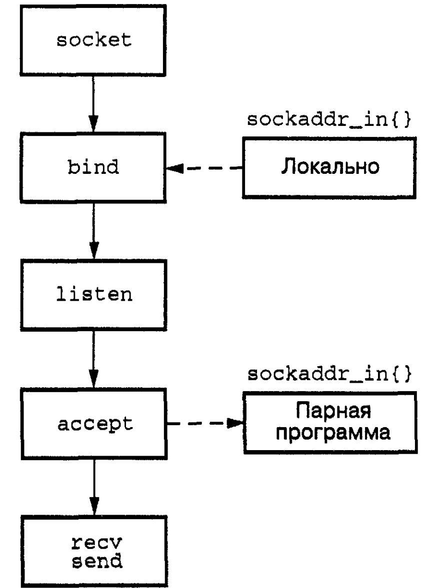


Рис. 1.3. Основные вызовы API сокетов для сервера

Традиционно значение параметра *backlog* не более пяти соединений, но в современных реализациях, которые должны поддерживать приложения с высокой нагрузкой, например, Web-сервера, оно может быть намного больше. Поэтому, чтобы выяснить его истинное значение, необходимо изучить документацию по конкретной системе. Если задать значение, большее максимально допустимого, то система уменьшит его, не сообщив об ошибке.

И последний вызов, который будет здесь рассмотрен, – это accept. Он служит для приема соединения, ожидающего во входной очереди. После того как соединение принято, его можно использовать для передачи данных, например, с помощью вызовов recv и send. В случае успеха accept возвращает дескриптор нового сокета, по которому и будет происходить обмен данными. Номер локального порта для этого сокета такой же, как и для прослушивающего сокета. Адрес интерфейса, на который поступил запрос о соединении, называется локальным. Адрес и номер порта клиента считаются удаленными.

Обратите внимание, что *оба* сокета имеют один и тот же номер локального порта. Это нормально, поскольку TCP-соединение полностью определяется четырьмя параметрами – локальным адресом, локальным портом, удаленным адресом и удаленным портом. Поскольку удаленные адрес и порт для этих двух сокетов различны, то ядро может отличить их друг от друга.

```
#include <sys/socket.h>  /* UNIX */
#include <winsock2.h>    /* Windows */

int accept( SOCKET s, struct sockaddr *addr, int *addrlen );
```

Возвращаемое значение: 0 – нормально, –1 (UNIX) или `INVALID_SOCKET` (Windows) – ошибка.

Параметр *s* – это дескриптор *прослушивающего* сокета. Как показано на рис. 1.3, `accept` возвращает адрес приложения на другом конце соединения в структуре `sockaddr_in`, на которую указывает параметр *addr*. Целому числу, на которое указывает параметр *addrlen*, ядро присваивает значение, равное длине этой структуры. Часто нет необходимости знать адрес клиентского приложения, поэтому в качестве *addr* и *addrlen* будет передаваться `NULL`.

В листинге 1.2 приведен простейший сервер. Эта программа также очень схематична, поскольку ее назначение – продемонстрировать структуру сервера и элементарные вызовы API сокетов, которые обязан выполнить любой сервер. Обратите внимание, что, как и в случае с клиентом на рис. 1.2, сервер следует потоку управления, показанному на рис. 1.3.

Листинг 1.2. Простой TCP-сервер

simples.c

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>

int main( void )
{
    struct sockaddr_in local;
    int s;
    int s1;
    int rc;
    char buf[ 1 ];

    local.sin_family = AF_INET;
    local.sin_port = htons( 7500 );
    local.sin_addr.s_addr = htonl( INADDR_ANY );

    s = socket( AF_INET, SOCK_STREAM, 0 );
    if ( s < 0 )
    {
        perror( "ошибка вызова socket" );
        exit( 1 );
```

```
20    }
21    rc = bind( s, ( struct sockaddr * )&local, sizeof( local ) );
22    if ( rc < 0 )
23    {
24       perror( "ошибка вызова bind" );
25       exit( 1 );
26    }
27    rc = listen( s, 5 );
28    if ( rc )
29    {
30       perror( "ошибка вызова listen" );
31       exit( 1 );
32    }
33    s1 = accept( s, NULL, NULL );
34    if ( s1 < 0 )
35    {
36       perror( "ошибка вызова accept" );
37       exit( 1 );
38    }
39    rc = recv( s1, buf, 1, 0 );
40    if ( rc <= 0 )
41    {
42       perror( "ошибка вызова recv" );
43       exit( 1 );
44    }
45    printf( "%c\n", buf[ 0 ] );
46    rc = send( s1, "2", 1, 0 );
47    if ( rc <= 0 )
48       perror( "ошибка вызова send" );
49    exit( 0 );
50 }
```

simples.c

Заполнение адресной структуры и получение сокета

12-20 Заполняем структуру `sockaddr_in`, записывая в ее поля известные адрес и номер порта, получаем сокет типа SOCK_STREAM, который и будет прослушивающим.

Привязка известного порта и вызов listen

21-32 Привязываем известные порт и адрес, записанные в структуру `local`, к полученному сокету. Затем вызываем `listen`, чтобы пометить сокет как прослушивающий.

Принятие соединения

33-39 Вызываем `accept` для приема новых соединений. Вызов `accept` блокирует выполнение программы до тех пор, пока не поступит запрос

на соединение, после чего возвращает новый сокет для этого соединения.

Обмен данными

39-49 Сначала читаем и печатаем байт со значением 1, полученный от клиента. Затем посылаем один байт со значением 2 назад клиенту и завершаем программу.

Теперь можно протестировать клиент и сервер, запустив сервер в одном окне, а клиент – в другом. Обратите внимание, что сервер должен быть запущен первым, иначе клиент аварийно завершится с сообщением `Connection refused` (В соединении отказано).

```
bsd: $ simplec
ошибка вызова connect: Connection refused
bsd: $
```

Ошибка произошла потому, что при попытке клиента установить соединение не было сервера, прослушивающего порт 7500.

Теперь следует поступить правильно, то есть запустить сервер до запуска клиента:

```
bsd: $ simples  |  bsd: $ simplec
1               |  2
bsd: $          |  bsd: $
```

Резюме

В этой главе приведен краткий обзор последующих глав и рассмотрены элементы API сокетов. Теперь можно перейти к более сложному материалу.

Глава 2. Основы

Совет 1. Различайте протоколы, требующие и не требующие установления логического соединения

Один из фундаментальных вопросов сетевого программирования – это различие между протоколами, требующими установления логического соединения (connection-oriented protocols), и протоколами, не требующими этого (connectionless protocols). Хотя ничего сложного в таком делении нет, но начинающие их часто путают. Частично проблема кроется в выборе слов. Очевидно, что два компьютера должны быть как-то «соединены», если необходимо наладить обмен данными между ними. Тогда что означает «отсутствие логического соединения»?

О наличии и отсутствии логического соединения говорят применительно к *протоколам*. Иными словами, речь идет о способе передачи данных по физическому носителю, а не о самом физическом носителе. Протоколы, требующие и не требующие логического соединения, могут одновременно разделять общий физический носитель; на практике обычно так и бывает.

Но если это деление не имеет ничего общего с физическим носителем, по которому передаются данные, то что же лежит в его основе? Главное различие в том, что в протоколах, не требующих соединения, каждый пакет передается независимо от остальных. Тогда как протоколы, устанавливающие соединение, поддерживают информацию о состоянии, которая позволяет следить за последовательностью пакетов.

При работе с протоколом, не требующим соединения, каждый пакет, именуемый *датаграммой*, адресуется и посылается приложением индивидуально (совет 30). С точки зрения протокола каждая датаграмма – это независимая единица, не имеющая ничего общего с другими датаграммами, которыми обмениваются приложения.

Примечание *Это не означает, что датаграммы независимы с точки зрения приложения. Если приложение реализует нечто более сложное, чем простой протокол запрос-ответ (клиент посылает серверу одиночный запрос и ожидает одиночного ответа на него), то, скорее всего, придется отслеживать состояние. Но суть в том, что приложение, а не протокол, отвечает за поддержание информации о состоянии. Пример сервера, который не требует установления соединения, но следит за последовательностью датаграмм, приведен в листинге 3.6.*

Обычно это означает, что клиент и сервер не ведут сложного диалога, – клиент посылает запрос, а сервер отвечает на него. Если позже клиент посылает новый запрос, то с точки зрения протокола это новая транзакция, не связанная с предыдущей.

Кроме того, протокол не обязательно надежен, то есть сеть предпримет все возможное для доставки каждой датаграммы, но нет гарантий, что ни одна не будет потеряна, задержана или доставлена не в том порядке.

С другой стороны, протоколы, требующие установления соединения, самостоятельно отслеживают состояние пакетов, поэтому они используются в приложениях, ведущих развитый диалог. Сохраняемая информация о состоянии позволяет протоколу обеспечить надежную доставку. Например, отправитель запоминает, когда и какие данные послал, но они еще не подтверждены. Если подтверждение не приходит в течение определенного времени, отправитель повторяет передачу. Получатель запоминает, какие данные уже принял, и отбрасывает пакеты-дубликаты. Если пакет поступает не в порядке очередности, то получатель может «придержать» его, пока не придут логически предшествующие пакеты.

У типичного протокола, требующего наличия соединения, есть три фазы. Сначала устанавливается соединение между двумя приложениями. Затем происходит обмен данными. И, наконец, когда оба приложения завершили обмен данными, соединение разрывается.

Обычно такой протокол сравнивают с телефонным разговором, а протокол, не требующий соединения, – с отправкой письма. Каждое письмо запечатывается в отдельный конверт, на котором пишется адрес. При этом все письма оказываются самостоятельными сущностями. Каждое письмо обрабатывается на почте независимо от других посланий двух данных корреспондентов. Почта не отслеживает историю переписки, то есть состояние последовательности писем. Кроме того, не гарантируется, что письма не затеряются, не задержатся и будут доставлены в правильном порядке. Это соответствует отправке датаграммы протоколом, не требующим установления соединения.

Примечание *Хаверлок [Haverlock 2000] отмечает, что более правильная аналогия – не письмо, а почтовая открытка, так как письмо с неправильным адресом возвращается отправителю, а почтовая открытка – никогда (как и в типичном протоколе, не требующем наличия соединения).*

А теперь посмотрим, что происходит, когда вы не посылаете письмо другу, а звоните по телефону. Для начала набираете его номер. Друг отвечает. Некоторое время вы разговариваете, потом прощаетесь и вешаете трубки. Так же обстоит дело и в протоколе, требующем соединения. В ходе процедуры установления соединения одна из сторон связывается с другой, стороны обмениваются «приветствиями» (на этом этапе они «договариваются» о тех параметрах и соглашениях, которым будут следовать далее), и соединение вступает в фазу обмена данными.

Во время телефонного разговора звонящий знает своего собеседника. И перед каждой фразой не нужно снова набирать номер телефона – соединение установлено.

Аналогично в фазе передачи данных протокола, требующего наличия соединения, не надо передавать свой адрес или адрес другой стороны. Эти адреса – часть информации о состоянии, хранящейся вместе с логическим соединением. Остается только посылать данные, не заботясь ни об адресации, ни о других деталях, связанных с протоколом.

Как и в разговоре по телефону, каждая сторона, заканчивая передачу данных, информирует об этом собеседника. Когда обе стороны договорились о завершении, они выполняют строго определенную процедуру разрыва соединения.

Примечание *Хотя указанная аналогия полезна, но она все же не точна. В телефонной сети устанавливается физическое соединение. А приводимое «соединение» целиком умозрительно, оно состоит лишь из хранящейся на обоих концах информации о состоянии. Чтобы должным образом понять это, подумайте, что произойдет, если хост на одном конце соединения аварийно остановится и начнет перезагружаться. Соединение все еще есть? По отношению к перезагрузившемуся хосту – конечно, нет. Все соединения установлены в его «прошлой жизни». Но для его бывшего «собеседника» соединение по-прежнему существует, так как у него все еще хранится информация о состоянии, и не произошло ничего такого, что сделало бы ее недействительной.*

В связи с многочисленными недостатками протоколов, не требующих соединения, возникает закономерный вопрос: зачем вообще нужен такой вид протоколов? Позже вы узнаете, что часто встречаются ситуации, когда для создания приложения использование именно такого протокола оправдано. Например, протокол без соединения может легко поддерживать связь одного хоста со многими и наоборот. Между тем протоколы, устанавливающие соединение, должны обычно организовать по одному соединению между каждой парой хостов. Важно то, что протоколы, не требующие наличия соединения, – это фундамент, на котором строятся более сложные протоколы. Рассмотрим набор протоколов TCP/IP. В совете 14 говорится, что TCP/IP – это четырехуровневый стек протоколов (рис. 2.1).

Приложение
TCP и UDP
IP
Интерфейс

Рис. 2.1 Упрощенное представление стека протоколов TCP/IP

Внизу стека находится интерфейсный уровень, который связан непосредственно с аппаратурой. Наверху располагаются такие приложения, как `telnet`, `ftp` и другие стандартные и пользовательские программы. Как видно из рис. 2.1, TCP и UDP построены поверх IP. Следовательно, IP – это фундамент, на котором возведено все здание TCP/IP. Но IP предоставляет лишь ненадежный сервис, не требующий установления соединения. Этот протокол принимает пакеты с вышерасположенных уровней, обертывает их в IP-пакет и направляет подходящему аппаратному интерфейсу для отправки в сеть. Послав пакет, IP, как и все протоколы, не устанавливающие соединения, не сохраняет информацию о нем.

В этой простоте и заключается главное достоинство протокола IP. Поскольку IP не делает никаких предположений о физической среде передачи данных, он может работать с любым носителем, способным передавать пакеты. Так, IP работает на простых последовательных линиях связи, в локальных сетях на базе технологий Ethernet и Token Ring, в глобальных сетях на основе протоколов X.25 и ATM (Asynchronous Transfer Mode – асинхронный режим передачи), в беспроводных сетях CDPD (Cellular Digital Packet Data – сотовая система передачи пакетов цифровых данных) и во многих других средах. Хотя эти технологии принципиально различны, с точки зрения IP они не отличаются друг от друга, поскольку способны передавать пакеты. Отсюда следует важнейший вывод: раз IP может работать в любой сети с коммутацией пакетов, то это относится и ко всему набору протоколов TCP/IP.

А теперь посмотрим, как протокол TCP пользуется этим простым сервисом, чтобы организовать надежный сервис с поддержкой логических соединений. Поскольку TCP-пакеты (они называются *сегментами*) посылаются в составе IP-датаграмм, у TCP нет информации, дойдут ли они до адреса, не говоря о возможности искажения данных или о доставке в правильном порядке. Чтобы обеспечить надежность, TCP добавляет к базовому IP-сервису три параметра. Во-первых, в TCP-сегмент включена контрольная сумма содержащихся в нем данных. Это позволяет в пункте назначения убедиться, что переданные данные не повреждены сетью во время транспортировки. Во-вторых, TCP присваивает каждому байту порядковый номер, так что даже если данные прибывают в пункт назначения не в том порядке, в котором были отправлены, то получатель сможет собрать из них исходное сообщение.

Примечание *Разумеется, TCP не передает порядковый номер вместе с каждым байтом. Просто в заголовке каждого TCP-сегмента хранится порядковый номер первого байта. Тогда порядковые номера остальных байтов можно вычислить.*

В-третьих, в TCP имеется механизм подтверждения и повторной передачи, который гарантирует, что каждый сегмент когда-то будет доставлен.

Из трех упомянутых выше добавлений механизм подтверждения/повторной передачи самый сложный, поэтому рассмотрим подробнее его работу.

Примечание *Здесь опускаются некоторые детали. Это обсуждение поверхностно затрагивает многие тонкости протокола TCP и их применение для обеспечения надежного и отказоустойчивого транспортного механизма. Более доступное и подробное изложение вы можете найти в RFC 793 [Postel 1981b] и RFC 1122 [Braden 1989], в книге [Stevens 1994]. В RFC 813 [Clark 1982] обсуждается механизм окон и подтверждений TCP.*

На каждом конце TCP-соединения поддерживается *окно приема*, представляющее собой диапазон порядковых номеров байтов, который получатель готов принять

от отправителя. Наименьшее значение, соответствующее левому краю окна, – это порядковый номер следующего ожидаемого байта. Наибольшее значение, соответствующее правому краю окна, – это порядковый номер последнего байта, для которого у TCP есть место в буфере. Использование окна приема (вместо посылки только номера следующего ожидаемого байта) повышает надежность протокола за счет предоставления средств управления потоком. Механизм управления потоком предотвращает переполнение буфера TCP.

Когда прибывает TCP-сегмент, все байты, порядковые номера которых оказываются вне окна приема, отбрасываются. Это касается как ранее принятых данных (с порядковым номерами левее окна приема), так и данных, для которых нет места в буфере (с порядковым номерами правее окна приема). Если первый допустимый байт в сегменте не является следующим ожидаемым, значит, сегмент прибыл не по порядку. В большинстве реализаций TCP такой сегмент помещается в очередь и находится в ней, пока не придут пропущенные данные. Если же номер первого допустимого байта совпадает со следующим ожидаемым, то данные становятся доступными для приложения, а порядковый номер следующего ожидаемого байта увеличивается на число байтов в сегменте. В этом случае считается, что окно сдвигается вправо на число принятых байтов. Наконец, TCP посылает отправителю подтверждение (сегмент ACK), содержащее порядковый номер следующего ожидаемого байта.

Например, на рис. 2.2а окно приема обведено пунктиром. Вы видите, что порядковый номер следующего ожидаемого байта равен 4, и TCP готов принять 9 байт (с 4 по 12). На рис. 2.2б показано окно приема после поступления байтов с номерами 4–7. Окно сдвинулось вправо на четыре номера, а в сегменте ACK, который пошлет TCP, номер следующего ожидаемого байта будет равен 8.



Рис. 2.2. Окно приема TCP

Теперь рассмотрим эту же ситуацию с точки зрения протокола TCP на посылающем конце. Помимо окна приема, TCP поддерживает также *окно передачи*, разделенное на две части. В одной из них расположены байты, которые уже отосланы, но еще не подтверждены, а в другой – байты, которые еще не отправлены. Предполагается, что на байты 1–3 уже пришло подтверждение, поэтому на рис. 2.3а изображено окно передачи, соответствующее окну приема на рис. 2.2а. На рис. 2.3б вы видите окно передачи после пересылки байтов 4–7, но до прихода подтверждения. TCP еще может послать байты 8–12, не дожидаясь подтверждения от получателя. После отправки байтов 4–7 TCP начинает отсчет тайм-аута ретрансмиссии (retransmission

timeout – RTO). Если до срабатывания таймера не пришло подтверждение на все четыре байта, TCP считает, что они потерялись, и посылает их повторно.

Примечание *Поскольку в многих реализациях не происходит отслеживания того, какие байты были посланы в конкретном сегменте, может случиться, что повторно переданный сегмент содержит больше байтов, чем первоначальный. Например, если байты 8 и 9 были посланы до срабатывания RTO-таймера, то такие реализации повторно передадут байты с 4 по 9.*

Обратите внимание, что срабатывание RTO-таймера не означает, что исходные данные не дошли до получателя. Например, может потеряться ACK-сегмент с подтверждением или исходный сегмент задержаться в сети на время, большее чем тайм-аут ретрансмиссии. Но ничего страшного в этом нет, так как если первоначально отправленные данные все-таки прибудут, то повторно переданные окажутся вне окна приема TCP и будут отброшены.

После получения подтверждения на байты 4–7 передающий TCP «забывает» про них и сдвигает окно передачи вправо, как показано на рис. 2.3в.

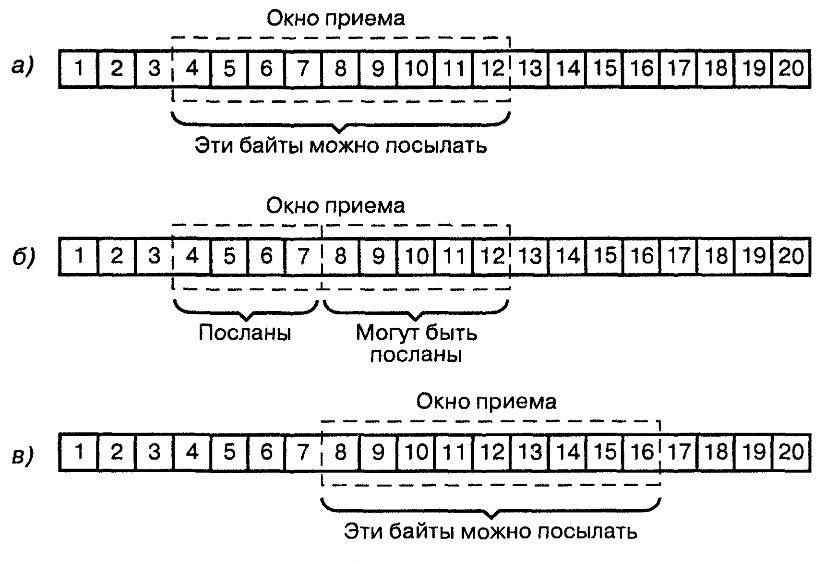


Рис. 2.3. Окно передачи TCP

TCP обеспечивает прикладного программиста надежным протоколом, требующим установления логических соединений. О таком протоколе рассказывается в совете 9.

С другой стороны, UDP предоставляет программисту ненадежный сервис, не требующий соединения. Фактически UDP добавляет лишь два параметра к протоколу IP, поверх которого он построен. Во-первых, необязательную контрольную сумму для обнаружения искаженных данных. Хотя у самого протокола IP тоже

есть контрольная сумма, но вычисляется она только для заголовка IP-пакета, поэтому TCP и UDP также включают контрольные суммы для защиты собственных заголовков и данных. Во-вторых, UDP добавляет к IP понятие порта.

Для отправки IP-датаграммы конкретному хосту используются IP-адреса, то есть адреса, которые обычно приводятся в стандартной десятичной нотации Internet (совет 2). Но по прибытии на хост назначения датаграмму еще необходимо доставить нужному приложению. Например, один UDP-пакет может быть предназначен для сервиса эхо-контроля, а другой – для сервиса «время дня». Порты как раз и дают способ направления данных нужному приложению (этот процесс называют *демультиплексированием*). С каждым TCP и UDP-сокетом ассоциирован номер порта. Приложение может явно указать этот номер путем обращения к системному вызову `bind` или поручить операционной системе выбор порта. Когда пакет прибывает, ядро «ищет» в списке сокетов тот, который ассоциирован с протоколом, парой адресов и парой номеров портов, указанных в пакете. Если сокет найден, то данные обрабатываются соответствующим протоколом (в примерах TCP или UDP) и передаются тем приложениям, которые этот сокет открыли.

Примечание *Если сокет открыт несколькими процессами или потоками (thread), то данные может считывать только один из них, остальным они будут недоступны.*

Возвращаясь к аналогии с телефонными переговорами и письмами, можно сказать, что сетевой адрес в TCP-соединении подобен номеру телефона офисной АТС, а номер порта – это добавочный номер конкретного телефона в офисе. Точно так же UDP-адрес можно представить как адрес многоквартирного дома, а номер порта – как отдельный почтовый ящик в его подъезде.

Резюме

В этом разделе обсуждены различия между протоколами, которые требуют и не требуют установления логического соединения. Вы узнали, что ненадежные протоколы, в которых происходит обмен датаграммами без установления соединения, – это фундамент, на котором строятся надежные протоколы на базе соединений. Попутно было кратко изложено, как надежный протокол TCP строится на основе ненадежного протокола IP.

Также отмечалось, что понятие «соединение» в TCP носит умозрительный характер. Оно состоит из хранящейся информации о состоянии на обоих концах; никакого «физического» соединения, как при телефонном разговоре, не существует.

Совет 2. Выясните, что такое подсети и CIDR

Длина IP-адреса (в версии IPv4) составляет 32 бита. Адреса принято записывать в десятичной нотации – каждый из четырех байт представляется одним десятичным числом, которые отделяются друг от друга точками. Так, адрес 0x11345678 записывается в виде 17.52.86.120. При записи адресов нужно учитывать, что в некоторых реализациях TCP/IP принято стандартное для языка C соглашение о том,

что числа, начинающиеся с нуля, записываются в восьмеричной системе. В таком случае 17.52.86.120 – это не то же самое, что 017.52.86.120. В первом примере адрес сети равен 17, а во втором – 15.

Классы адресов

По традиции все IP-адреса подразделены на пять классов, показанных на рис. 2.4.

Адреса класса D используются для группового вещания, а класс E зарезервирован для будущих расширений. Остальные классы – A, B и C – предназначены для адресации отдельных сетей и хостов.

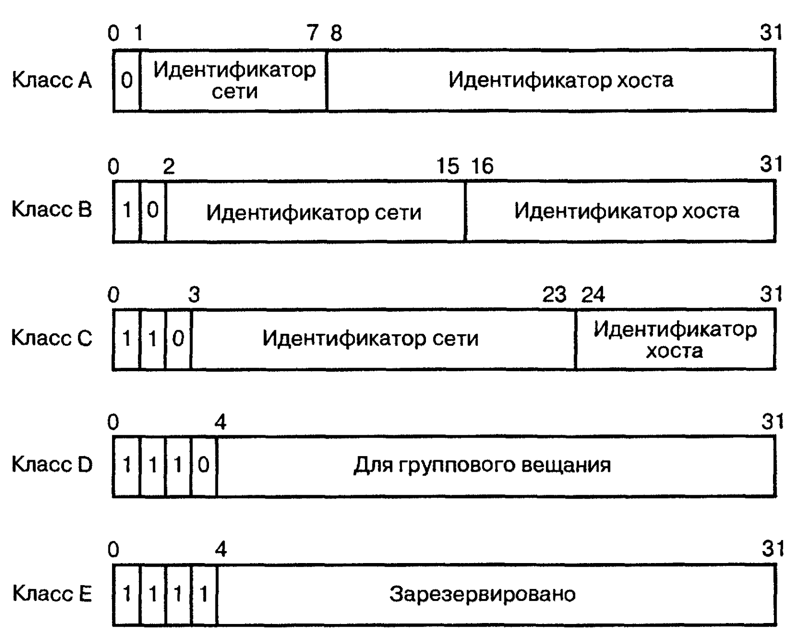


Рис. 2.4. Классы IP-адресов

Класс адреса определяется числом начальных единичных битов. У адресов класса A вообще нет бита 1 в начале, у адресов класса B – один такой бит, у адресов класса C – два и т.д. Идентификация класса адреса чрезвычайно важна, поскольку от этого зависит интерпретация остальных битов адреса.

Остальные биты любого адреса классов A, B и C разделены на две группы Первая часть любого адреса представляет собой идентификатор сети, вторая – идентификатор хоста внутри этой сети.

Примечание *Биты идентификации класса также считаются частью идентификатора сети. Так, 130.50.10.200 – это адрес класса B, в котором идентификатор сети равен 0x8232.*

Смысл разбивки адресного пространства на классы в том, чтобы обеспечить необходимую гибкость, не теряя адресов Например, класс A позволяет адресовать сети с огромным (16777214) количеством хостов.

Примечание *Существует 2^{24}, или 16777216 возможных идентификаторов хостов, но адрес 0 и адрес, состоящий из одних единиц, имеют специальный смысл. Адрес из одних единиц – это широковещательный адрес. IP-датаграммы, посланные по этому адресу, доставляются всем хостам в сети. Адрес 0 означает «этот хост» и используется хостом как адрес источника, которому в ходе процедуры начальной загрузки необходимо определить свой истинный сетевой адрес. Поэтому число хостов в сети всегда равно $2^n - 2$, где n – число бит в части адреса, относящейся к хосту.*

Поскольку в адресах класса A под идентификатор сети отводятся 7 бит, то всего существует 128 сетей класса A.

Примечание *Как и в случае идентификаторов хостов, два из этих адресов зарезервированы. Адрес 0 означает «эта сеть» и, аналогично хосту 0, используется для определения адреса сети в ходе начальной загрузки. Адрес 127 – это адрес «собственной» сети хоста. Датаграммы, адресованные сети 127, не должны покидать хост-отправитель. Часто этот адрес называют «возвратным» (loopback) адресом, поскольку отправленные по нему датаграммы «возвращаются» на тот же самый хост.*

На другом полюсе располагаются сети класса C. Их очень много, но в каждой может быть не более 254 хостов. Таким образом, адреса класса A предназначены для немногих гигантских сетей с миллионами хостов, тогда как адреса класса C – для миллионов сетей с небольшим количеством хостов.

В табл. 2.1 показано, сколько сетей и хостов может существовать в каждом классе, а также диапазоны допустимых адресов. Будем считать, что сеть 127 принадлежит классу A, хотя на самом деле она, конечно, недоступна для адресации.

Таблица 2.1. Число сетей, хостов и диапазоны адресов для классов A, B и C

Класс	Сети	Хосты	Диапазон адресов
A	127	16 777 214	0.0.0.1–127.255.255.255
B	16 384	65 534	128.0.0.0–191.255.255.255
C	2 097 252	254	192.0.0.0–223.255.255.255

Первоначально проектировщики набора протоколов TCP/IP полагали, что сети будут исчисляться сотнями, а хосты – тысячами.

Примечание *В действительности, как отмечается в работе [Huitema 1995], в исходном проекте фигурировали только адреса, которые теперь относятся к классу A. Подразделение на три класса было сделано позже, чтобы иметь более 256 сетей.*

Появление дешевых, повсеместно применяемых персональных компьютеров привело к значительному росту числа сетей и хостов. Нынешний размер Internet намного превосходит ожидания его проектировщиков.

Такой рост выявил некоторые недостатки классов адресов. Прежде всего, число хостов в классах А и В слишком велико. Вспомним, что идентификатор сети, как предполагалось, относится к физической сети, например локальной. Но никто не станет строить физическую сеть из 65000 хостов, не говоря уже о 16000000. Вместо этого большие сети разбиваются на сегменты, взаимосвязанные маршрутизаторами.

В качестве простого примера рассмотрим два сегмента сети, изображенной на рис. 2.5.

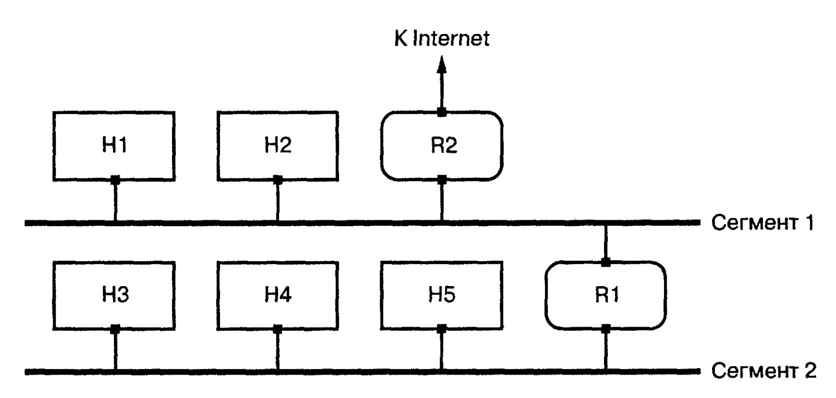


Рис. 2.5. Сеть из двух сегментов

Если хосту H1 нужно обратиться к хосту H2, то он получает физический адрес, соответствующий IP-адресу H2 (используя для этого метод, свойственный данной реализации физической сети), и помещает датаграмму «на провод».

А если хосту H1 необходимо обратиться к хосту H3? Напрямую послать датаграмму невозможно, даже если известен физический адрес получателя, поскольку H1 и H3 находятся в разных сетях. Поэтому H1 должен отправить датаграмму через маршрутизатор R1. Если у двух сегментов разные идентификаторы сетей, то H1 по своей маршрутной таблице определяет, что пакеты, адресованные сегменту 2, обрабатываются маршрутизатором R1, и отправляет ему датаграмму в предположении, что тот переправит ее хосту H3.

Итак, можно назначить двум сегментам различные идентификаторы сети. Но есть и другие решения в рамках системы адресных классов. Во-первых, маршрутная таблица хоста H1 может содержать по одному элементу для каждого хоста в сегменте 2, который определит следующего получателя на пути к этому хосту – R1. Такая же таблица должна размещаться на каждом хосте в сегменте 1. Аналогичные таблицы, описывающие достижимость хостов из сегмента 1, следует поместить на каждом хосте из сегмента 2. Очевидно, такое решение плохо масштабируется при значительном количестве хостов. Кроме того, маршрутные таблицы придется вести вручную, что очень скоро станет непосильной задачей для администратора. Поэтому на практике такое решение почти никогда не применяется.

Во-вторых, можно реализовать ARP-прокси (proxy ARP) таким образом, чтобы R1 казался для хостов из сегмента 1 одновременно H3, H4 и H5, а для хостов из сегмента 2 – H1, H2 и R2.

Примечание *Агента ARP в англоязычной литературе еще называют promiscuous ARP (пропускающий ARP) или ARP hack (трюк ARP).*

Это решение годится только в случае, когда в физической сети используется протокол ARP (Address Resolution Protocol – протокол разрешения адресов) для отображения IP-адресов на физические адреса. В соответствии с ARP хост, которому нужно получить физический адрес, согласующийся с некоторым IP-адресом, должен послать широковещательное сообщение с просьбой хосту, обладающему данным IP-адресом, выслать свой физический адрес. ARP-запрос получают все хосты в сети, но отвечает только тот, IP-адрес которого совпадает с запрошенным.

Если применяется агент ARP, то в случае, когда хосту H1 необходимо послать IP-датаграмму H3, физический адрес которого неизвестен, он посылает ARP-запрос физического адреса H3. Но H3 этот запрос не получит, поскольку находится в другой сети. Поэтому на запрос отвечает его агент – R1, сообщая свой собственный адрес. Когда R1 получает датаграмму, адресованную H3, он переправляет ее конечному адресату. Все происходит так, будто H3 и H1 находятся в одной сети.

Как уже отмечалось, агент ARP может работать только в сетях, которые используют протокол ARP и к тому же имеют сравнительно простую топологию. Подумайте, что случится при наличии нескольких маршрутизаторов, соединяющих сегменты 1 и 2.

Из вышесказанного следует, что общий способ организовать сети с несколькими сегментами – это назначить каждому сегменту свой идентификатор сети. Но у этого решения есть недостатки. Во-первых, при этом возможна потеря многих адресов в каждой сети. Так, если у любого сегмента сети имеется свой адрес класса B, то большая часть IP-адресов просто не будет использоваться.

Во-вторых, маршрутная таблица любого узла, который направляет датаграммы напрямую в комбинированную сеть, должна содержать по одной записи для каждого сегмента. В указанном примере это не так страшно. Но вообразите сеть из нескольких сотен сегментов, а таких сетей может быть много. Понятно, что размеры маршрутных таблиц станут громадными.

Примечание *Эта проблема более серьезна, чем может показаться на первый взгляд. Объем памяти маршрутизаторов обычно ограничен, и нередко маршрутные таблицы размещаются в памяти специального назначения на сетевых картах. Реальные примеры отказа маршрутизаторов из-за роста маршрутных таблиц рассматриваются в работе [Huitema 1995].*

Обратите внимание, что эти проблемы не возникают при наличии хотя бы одного идентификатора сети. IP-адреса не остаются неиспользованными, поскольку

при потребности в новых хостах можно всегда добавить новый сегмент. С другой стороны, так как имеется лишь один идентификатор сети, в любой маршрутной таблице необходима всего одна запись для отправки датаграмм любому хосту в этой сети.

Подсети

Мне хотелось найти решение, сочетающее два достоинства: во-первых, небольшие маршрутные таблицы и эффективное использование адресного пространства, обеспечиваемые единым идентификатором сети, во-вторых, простота маршрутизации, характерная для сетей, имеющих сегменты с разными идентификаторами сети. Желательно, чтобы внешние хосты «видели» только одну сеть, а внутренние – несколько сетей, по одной для каждого сегмента.

Это достигается с помощью механизма *подсетей*. Идея очень проста. Поскольку внешние хосты для принятия решения о выборе маршрута используют только идентификатор сети, администратор может распределять идентификаторы хостов по своему усмотрению. Таким образом, идентификатор хоста – это закрытая структура, не имеющая вне данной сети интерпретации.

Разделение на подсети осуществляется по следующему принципу. Одна часть идентификатора хоста служит для определения сегмента (то есть подсети), в состав которого входит хост, а другая – для идентификации конкретного хоста. Рассмотрим, например, сеть класса В с адресом 190.50.0.0. Можно считать, что третий байт адреса – это идентификатор подсети, а четвертый байт – номер хоста в этой подсети. На рис. 2.6а приведена структура адреса с точки зрения внешнего компьютера. Идентификатор хоста – это поле с заранее неизвестной структурой. На рис. 2.6б показано, как эта структура выглядит изнутри сети. Вы видите, что она состоит из идентификатора подсети и номера хоста.

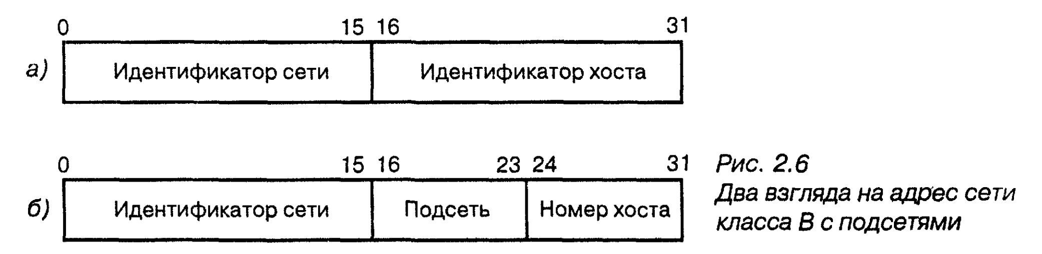


Рис. 2.6 Два взгляда на адрес сети класса В с подсетями

В приведенном примере взят адрес класса В, и поле номера хоста выделено по границе байта. Но это необязательно. На подсети можно разбивать сети классов А, В и С и часто не по границе байта. С каждой подсетью ассоциируется *маска подсети*, которой определяется, какая часть адреса отведена под идентификаторы сети и подсети, а какая – под номер хоста. Так, маска подсети для примера, показанного на рис. 2.6б, будет 0xffffff00. В основном маска записывается в десятичной нотации (255.255.255.0), но если разбивка проходит не по границе байта, то удобнее первая форма.

Примечание *Обратите внимание, что, хотя говорится о маске подсети, фактически она выделяет части, относящиеся как к сети, так и к подсети, то есть все, кроме номера хоста.*

Предположим, что для идентификатора подсети отведено 10 бит, а для номера хоста – 6 бит. Тогда маска подсети будет 255.255.255.192 (0xffffffc0). Как следует из рис. 2.7, в результате наложения этой маски на адрес 190.50.7.75 получается номер сети/подсети, равный 190.70.7.64.

Для проверки убедитесь, что адрес 190.50.7.75 принадлежит хосту 11 в подсети 29 сети 190.50.0.0. Важно не забывать, что эта интерпретация имеет смысл только внутри сети. Для внешнего мира адрес интерпретируется как хост 1867 в сети 190.50.0.0.

Теперь следует выяснить, как маршрутизаторы на рис. 2.5 могут воспользоваться структурой идентификатора хоста для рассылки датаграмм внутри сети. Предположим, что есть сеть класса B с адресом 190.5.0.0 и маска подсети равна 255.255.255.0. Такая структура показана на рис. 2.6б.

	190	50	7	75
IP-адрес	10111110	00110010	00000111	01001011
Маска подсети	11111111	11111111	11111111	11000000
Сеть/подсеть	10111110	00110010	00000111	01000000
	190	50	7	64

Рис. 2.7. Наложение маски подсети с помощью операции AND для выделения сетевой части IP-адреса

На рис. 2.8 первому сегменту назначен идентификатор подсети 1, а второму – идентификатор подсети 2. Рядом с сетевым интерфейсом каждого хоста указан его IP-адрес. Обратите внимание, что третий байт каждого адреса – это номер подсети, которой принадлежит интерфейс. Однако внешнему компьютеру эта интерпретация неизвестна.

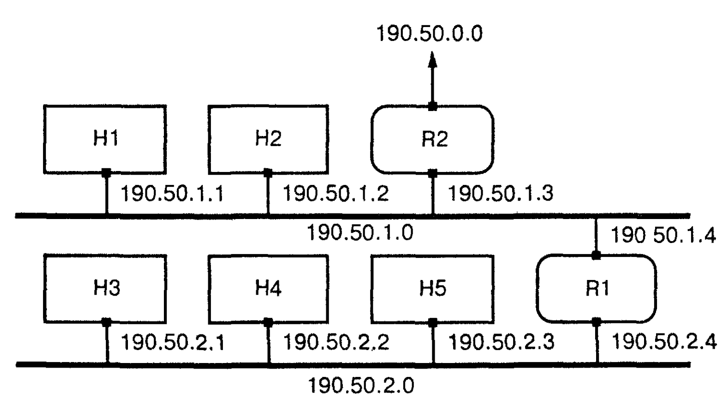


Рис. 2.8. Сеть с подсетями

Возвращаясь к вышесказанному, следует выяснить, что происходит, когда хосту H1 нужно обратиться к хосту H3. H1 берет адрес H3 (190.50.2.1) и накладывает на него маску подсети (255.255.255.0), получая в результате 190.5.2.0. Поскольку H1 находится в подсети 190.5.1.0, то H3 напрямую недоступен, поэтому он сверяется со своей маршрутной таблицей и обнаруживает, что следующий адрес на пути к H3 – это R1.

Примечание *Во многих реализациях эти два шага объединены за счет помещения в маршрутную таблицу обеих подсетей. При поиске маршрута IP выявляет одно из двух: либо целевая сеть доступна непосредственно, либо датаграмму надо отослать промежуточному маршрутизатору.*

Затем H1 отображает IP-адрес R1 на его физический адрес (например, с помощью протокола ARP) и посылает R1 датаграмму. R1 ищет адрес назначения в своей маршрутной таблице, пользуясь той же маской подсети, и определяет местонахождение H3 в подсети, соединенной с его интерфейсом 190.50.2.4. После чего R1 доставляет датаграмму хосту H3, получив предварительно его физический адрес по IP-адресу, – для этого достаточно передать датаграмму сетевому интерфейсу 190.50.2.4.

А теперь предположим, что H1 необходимо отправить датаграмму H2. При наложении маски подсети на адрес H2 (190.5.1.2) получается 190.50.1.0, то есть та же подсеть, в которой находится сам хост H1. Поэтому H1 нужно только получить физический адрес H2 и отправить ему датаграмму напрямую.

Далее разберемся, что происходит, когда хосту E из внешней сети нужно отправить датаграмму H3. Поскольку 190.50.2.1 – адрес класса B, то маршрутизатору на границе сети хоста E известно, что H3 находится в сети 190.50.0.0. Так как шлюзом в эту сеть является R2, рано или поздно датаграмма от хоста E дойдет до этого маршрутизатора. С этого момента все совершается так же, как при отправке датаграммы хостом H1: R2 накладывает маску, выделяет адрес подсети 190.50.2.0, определяет R1 в качестве следующего узла на пути к H3 и посылает R1 датаграмму, которую тот переправляет H3. Заметьте, что хосту E неизвестна внутренняя топология сети 190.50.0.0. Он просто посылает датаграмму шлюзу R2. Только R2 и другие хосты внутри сети определяют существование подсетей и маршруты доступа к ним.

Важный момент, который нужно помнить, – маска подсети ассоциируется с сетевым интерфейсом и, следовательно, с записью в маршрутной таблице. Это означает, что разные подсети в принципе могут иметь разные маски.

Предположим, что адрес класса B 190.50.0.0 принадлежит университетской сети, а каждому факультету выделена подсеть с маской 255.255.255.0 (на рис. 2.8 показана только часть всей сети). Администратор факультета информатики, которому назначена подсеть 5, решает выделить один сегмент сети Ethernet компьютерному классу, а другой – всем остальным факультетским компьютерам. Он мог бы потребовать у администрации университета еще один номер подсети, но в компьютерном классе всего несколько машин, так что нет смысла выделять ему адресное пространство, эквивалентное целой подсети класса C. Вместо этого он предпочел разбить свою подсеть на два сегмента, то есть создать подсеть внутри подсети.

Для этого он увеличивает длину поля подсети до 10 бит и использует маску 255.255.255.192. В результате структура адреса выглядит, как показано на рис. 2.9.

Старшие 8 бит идентификатора подсети всегда равны 0000 0101 (5), поскольку основная сеть адресует всю подсеть как подсеть 5. Биты X и Y определяют, какой Ethernet-сегмент внутри подсети 190.50.5.0 адресуется. Из рис. 2.10 видно, что если XY = 10, то адресуется подсеть в компьютерном классе, а если XY = 01 – оставшаяся часть сети. Частично топология подсети 190.50.5.0 изображена на рис. 2.10.

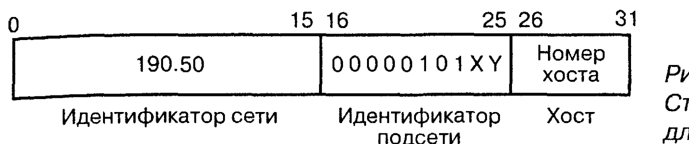


Рис. 2.9
Структура адреса для подсети 190.50.5.0

В верхнем сегменте (подсеть 190.50.1.0) на рис. 2.10 расположен маршрутизатор R2, обеспечивающий выход во внешний мир, такой же, как на рис. 2.8. Подсеть 190.50.2.0 здесь не показана. Средний сегмент (подсеть 190.50.5.128) – это локальная сеть Ethernet в компьютерном классе. Нижний сегмент (подсеть 190.50.5.64) – это сеть Ethernet, объединяющая остальные факультетские компьютеры. Для упрощения номер хоста каждой машины один и тот же для всех ее сетевых интерфейсов и совпадает с числом внутри прямоугольника, представляющего хост или маршрутизатор.

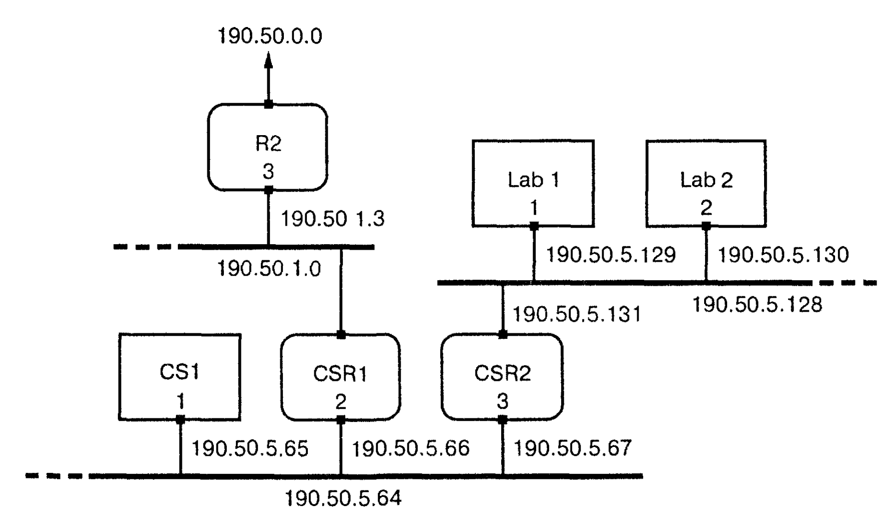


Рис. 2.10. Подсеть внутри подсети

Маска подсети для интерфейсов, подсоединенных к подсетям 190.50.5.64 и 190.50.5.128, равна 255.255.255.192, а к подсети 190.50.1.0 – 255.255.255.0.

Эта ситуация в точности аналогична предыдущей, которая рассматривалась для рис. 2.8. Так же, как хостам вне сети 190.50.0.0 неизвестно то, что третий байт адреса определяет подсеть, так и хосты в сети 190.50.0.0, но вне подсети 190.50.5.0, не могут определить, что первые два бита четвертого байта задают подсеть подсети 190.50.5.0.

Теперь кратко остановимся на широковещательных адресах. При использовании подсетей существует четыре типа таких адресов для вещания: ограниченный, на сеть, на подсеть и на все подсети.

Ограниченное вещание

Адрес для ограниченного вещания – 255.255.255.255. Вещание называется *ограниченным*, поскольку датаграммы, посланные на этот адрес, не уходят дальше

маршрутизатора. Они ограничены локальным кабелем. Такое широковещание применяется, главным образом, во время начальной загрузки, если хосту неизвестен свой IP-адрес или маска своей подсети.

Процесс передачи широковещательной датаграммы хостом, имеющим несколько сетевых интерфейсов, зависит от реализации. Во многих реализациях датаграмма отправляется только по одному интерфейсу. Чтобы приложение отправило широковещательную датаграмму по нескольким интерфейсам, ему необходимо узнать у операционной системы, какие интерфейсы сконфигурированы для поддержки широковещания.

Вещание на сеть

В адресе для вещания на сеть идентификатор сети определяет адрес этой сети, а идентификатор хоста состоит из одних единиц. Например, для вещания на сеть 190.50.0.0 используется адрес 190.50.255.255. Датаграммы, посылаемые на такой адрес, доставляются всем хостам указанной сети.

Требования к машрутизаторам (RFC 1812) [Baker 1995] предусматривают по умолчанию пропуск маршрутизатором сообщений, вещаемых на сеть, но эту возможность можно отключить. Во избежание атак типа «отказ от обслуживания» (denial of service), которые используют возможности, предоставляемые направленным широковещанием, во многих маршрутизаторах пропуск таких датаграмм, скорее всего, будет заблокирован.

Вещание на подсеть

В адресе для вещания на все подсети идентификаторы сети и подсети определяют соответствующие адреса, а идентификатор хоста состоит из одних единиц. Не зная маски подсети, невозможно определить, является ли данный адрес адресом для вещания на подсеть. Например, адрес 190.50.1.255 можно трактовать как адрес для вещания на подсеть только при условии, если маршрутизатор имеет информацию, что маска подсети равна 255.255.255.0. Если же известно, что маска подсети равна 255.255.0.0, то это адрес не считается широковещательным.

При использовании бесклассовой междоменной маршрутизации (CIDR), которая будет рассмотрена ниже, широковещательный адрес этого типа такой же, как и адрес вещания на сеть; RFC 1812 предлагает трактовать их одинаково.

Вещание на все подсети

В адресе для вещания на все подсети задан идентификатор сети, а адреса подсети и хоста состоят из одних единиц. Как и при вещании на подсеть, для опознания такого адреса необходимо знать маску подсети.

К сожалению, применение адреса для вещания на все подсети сопряжено с некоторыми проблемами, поэтому этот режим не внедрен. При использовании CIDR этот вид широковещания не нужен и, по RFC 1812, «отправлен на свалку истории».

Ни один из описанных широковещательных адресов нельзя использовать в качестве адреса источника IP-датаграммы. И, наконец, следует отметить, что в некоторых ранних реализациях TCP/IP, например в системе 4.2BSD, для выделения широковещательного адреса в поле идентификатора хоста ставились не единицы, а нули.

Бесклассовая междоменная маршрутизация – CIDR

Теперь вам известно, как организация подсетей решает одну из проблем, связанных с классами адресов: переполнение маршрутных таблиц. Хотя и в меньшей степени, подсети все же позволяют справиться и с проблемой истощения IP-адресов за счет лучшего использования пула идентификаторов хостов в пределах одной сети.

Еще одна серьезная проблема – это недостаток сетей класса B. Как показано на рис. 2.5, существует менее 17000 таких сетей. Поскольку большинство средних и крупных организаций нуждается в количестве IP-адресов, превышающем возможности сети класса C, им выделяется идентификатор сети класса B.

В условиях дефицита сетей класса B организациям приходилось выделять блоки адресов сетей класса C, но при этом вновь возникает проблема, которую пытались решить с помощью подсетей, – растут маршрутные таблицы.

Бесклассовая междоменная маршрутизация (CIDR) решает эту проблему, вывернув принцип организации подсетей «наизнанку». Вместо увеличения CIDR уменьшает длину идентификатора сети в IP-адресе.

Предположим, некоторой организации нужно 1000 IP-адресов. Ей выделяют четыре соседних идентификатора сетей класса C с общим префиксом от 200.10.4.0 до 200.10.7.0. Первые 22 бита этих идентификаторов одинаковы и представляют номер агрегированной сети, в данном случае 200.10.4.0. Как и для подсетей, для идентификации сетевой части IP-адреса используется маска сети. В приведенном здесь примере она равна 255.255.252.0 (0xfffffc00).

Но в отличие от подсетей эта маска сети не расширяет сетевую часть адреса, а укорачивает ее. Поэтому CIDR называют также *суперсетями*. Кроме того, маска сети в отличие от маски подсети экспортируется во внешний мир. Она становится частью любой записи маршрутной таблицы, ссылающейся на данную сеть.

Допустим, внешнему маршрутизатору R надо переправить датаграмму по адресу 200.10.5.33, который принадлежит одному из хостов в агрегированной сети. Он просматривает записи в своей маршрутной таблице, в каждой из которых хранится маска сети, и сравнивает замаскированную часть адреса 200.10.5.33 с хранящимся в записи значением. Если в таблице есть запись для сети, то в ней будет храниться адрес 200.10.4.0 и маска сети 255.255.252.0. Когда выполняется операция побитового AND между адресом 200.10.5.33 и этой маской, получается значение 200.10.4.0. Это значение совпадает с хранящимся в записи номером подсети, так что маршрутизатору известно, что именно по этому адресу следует переправить датаграмму.

Если возникает неоднозначность, то берется самое длинное соответствие. Например, в маршрутной таблице может быть также запись с адресом 200.10.0.0

и маской сети 255.255.0.0. Эта запись также соответствует адресу 200.10.5.33, но поскольку для нее совпадают только 16 бит, а не 22, как в первом случае, то предпочтение отдается первой записи.

Примечание *Может случиться так, что Internet сервис-провайдер (ISP) «владеет» всеми IP-адресами с префиксом 200.10. В соответствии со второй из рассмотренных выше записей маршрутизатор отправил бы этому провайдеру все датаграммы, адрес назначения которых начинается с 200.10. Тогда провайдер смог бы указать более точный маршрут, чтобы избежать лишних звеньев в маршруте или по какой-то иной причине.*

В действительности механизм CIDR более общий. Он называется «бесклассовым», так как понятие «класса» в нем полностью отсутствует. Таким образом, каждая запись в маршрутной таблице содержит маску сети, определяющую сетевую часть IP-адреса. Если принять, что адрес принадлежит некоторому классу, то эта маска может укоротить или удлинить сетевую часть адреса. Но поскольку в CIDR понятия «класса» нет, то можно считать, что сетевая маска выделяет сетевую часть адреса без изменения ее длины.

В действительности, маска – это всего лишь число, называемое *префиксом*, которое определяет число бит в сетевой части адреса. Например, для вышеупомянутой агрегированной сети префикс равен 22, и адрес этой сети следовало бы записать как 200.10.4.0/22, где /22 обозначает префикс. С этой точки зрения адресацию на основе классов можно считать частным случаем CIDR, когда имеется всего четыре (или пять) возможных префиксов, закодированных в старших битах адреса.

Гибкость, с которой CIDR позволяет задавать размер адреса сети, позволяет эффективно распределять IP-адреса блоками, размер которых оптимально соответствует потребностям сети. Вы уже видели, как можно использовать CIDR для агрегирования нескольких сетей класса С в одну большую сеть. А для организации маленькой сети из нескольких хостов можно выделить лишь часть адресов сети класса С. Например, сервис-провайдер выделяет небольшой компании с единственной ЛВС адрес сети 200.50.17.128/26. В такой сети может существовать до 62 хостов (2^6–2).

В RFC 1518 [Rekhter и Li 1993] при обсуждении вопроса об агрегировании адресов и его влиянии на размер маршрутных таблиц рекомендуется выделять префиксы IP-адресов (то есть сетевые части адреса) иерархически.

Примечание *Иерархическое агрегирование адресов можно сравнить с иерархической файловой системой вроде тех, что используются в UNIX и Windows. Так же, как каталог верхнего уровня содержит информацию о своих подкаталогах, но не имеет сведений о находящихся в них файлах, доменам маршрутизации верхнего уровня известно лишь о промежуточных доменах, а не о конкретных сетях внутри них. Предположим, что региональный провайдер обеспечивает весь трафик для префикса 200/8, а к нему подключены три локальных провайдера с префиксами*

200.1/16, 200.2/16 и 200.3/16. У каждого провайдера есть несколько клиентов, которым выделены части располагаемого адресного пространства (200.1.5/24 и т.д.). Маршрутизаторы, внешние по отношению к региональному провайдеру, должны хранить в своих таблицах только одну запись – 200/8. Этого достаточно для достижения любого хоста в данном диапазоне адресов. Решения о выборе маршрута можно принимать, даже не зная о разбиении адресного пространства 200/8. Маршрутизатор регионального провайдера должен хранить в своей таблице только три записи: по одной для каждого локального провайдера. На самом нижнем уровне локальный провайдер хранит записи для каждого своего клиента. Этот простой пример позволяет видеть суть агрегирования.

Почитать RFC 1518 очень полезно, поскольку в этом документе демонстрируются преимущества использования CIDR. В RFC 1519 [Fuller et al. 1993] описаны CIDR и ее логическое обоснование, а также приведены подробный анализ затрат, связанных с CIDR, и некоторые изменения, которые придется внести в протоколы междоменной маршрутизации.

Текущее состояние организации подсетей и CIDR

Подсети в том виде, в каком они описаны в RFC 950 [Mogul and Postel 1985], – это часть Стандартного протокола (Std. 5). Это означает, что каждый хост, на котором установлен стек TCP/IP, обязан поддерживать подсети.

CIDR (RFC 1517 [Hinden 1993], RFC 1518, RFC 1519) – часть предложений к стандартному протоколу, и потому не является обязательной. Тем не менее CIDR применяется в Internet почти повсеместно, и все новые адреса выделяются этим способом. Группа по перспективным разработкам в Internet (IESG – Internet Engineering Steering Group) выбрала CIDR как промежуточное временное решение проблемы роста маршрутных таблиц.

В перспективе обе проблемы – исчерпания адресов и роста маршрутных таблиц – предполагается решать с помощью версии 6 протокола IP. IPv6 имеет большее адресное пространство (128 бит) и изначально поддерживает иерархию. Такое адресное пространство (включая 64 бита для идентификатора интерфейса) гарантирует, что вскоре IP-адресов будет достаточно. Иерархия IPv6-адресов позволяет держать размер маршрутных таблиц в разумных пределах.

Резюме

В этом разделе рассмотрены подсети и бесклассовая междоменная маршрутизация (CIDR). Вы узнали, как они применяются для решения двух проблем, свойственных адресации на основе классов. Подсети позволяют предотвратить рост маршрутных таблиц, обеспечивая в то же время гибкую адресацию. CIDR служит для эффективного выделения IP-адресов и способствует их иерархическому назначению.

Совет 3. Разберитесь, что такое частные адреса и NAT

Раньше, когда доступ в Internet еще не был повсеместно распространен, организации выбирали произвольный блок IP-адресов для своих сетей. Считалось, что сеть не подключена и «никогда не будет подключена» к внешним сетям, поэтому выбор IP-адресов не имеет значения. Но жизнь не стоит на месте, и в настоящее время очень мало сетей, которые не имеют выхода в Internet.

Теперь необязательно выбирать для частной сети произвольный блок IP-адресов. В RFC 1918 [Rekhter, Moskowitz et al. 1996] специфицированы три блока адресов, которые не будут выделяться:

- 10.0.0.0–10.255.255.255 (префикс 10/8);
- 172.16.0.0–172.31.255.255 (префикс 172.16/12);
- 192.168.0.0–192.168.255.255 (префикс 192.168/16).

Если использовать для своей сети один из этих блоков, то любой хост сможет обратиться к другому хосту в этой же сети, не опасаясь конфликта с глобально выделенным IP-адресом. Разумеется, пока сеть не имеет выхода во внешние сети, выбор адресов не имеет значения. Но почему бы сразу не воспользоваться одним из блоков частных адресов и не застраховаться тем самым от неприятностей, которые могут произойти, когда внешний выход все-таки появится?

Что случится, когда сеть получит внешний выход? Как хост с частным IP-адресом сможет общаться с другим хостом в Internet или другой внешней сети? Самый распространенный ответ – нужно воспользоваться *преобразованием сетевых адресов* (Network Address Translation – NAT). Есть несколько типов устройств, поддерживающих NAT. Среди них маршрутизаторы, межсетевые экраны (firewalls) и автономные устройства с поддержкой NAT. Принцип работы NAT заключается в преобразовании между частными сетевыми адресами и одним или несколькими глобально выделенными IP-адресами. Большинство устройств с поддержкой NAT можно сконфигурировать в трех режимах:

- статический. Адреса всех или некоторых хостов в частной сети отображаются на один и тот же фиксированный, глобально выделенный адрес;
- выбор из пула. Устройство с поддержкой NAT имеет пул глобально выделенных IP-адресов и динамически назначает один из них хосту, которому нужно связаться с хостом во внешней сети;
- PAT, или *преобразование адресов портов* (port address translation). Этот метод применяется, когда есть единственный глобально выделенный адрес (рис. 2.11). При этом каждый частный адрес отображается на один и тот же внешний адрес, но номер порта исходящего пакета заменяется уникальным значением, которое в дальнейшем используется для ассоциирования входящих пакетов с частным сетевым адресом.

На рис. 2.11 представлена небольшая сеть с тремя хостами, для которой используется блок адресов 10/8. Имеется также маршрутизатор, помеченный NAT, у которого есть адрес в частной сети и адрес в Internet.

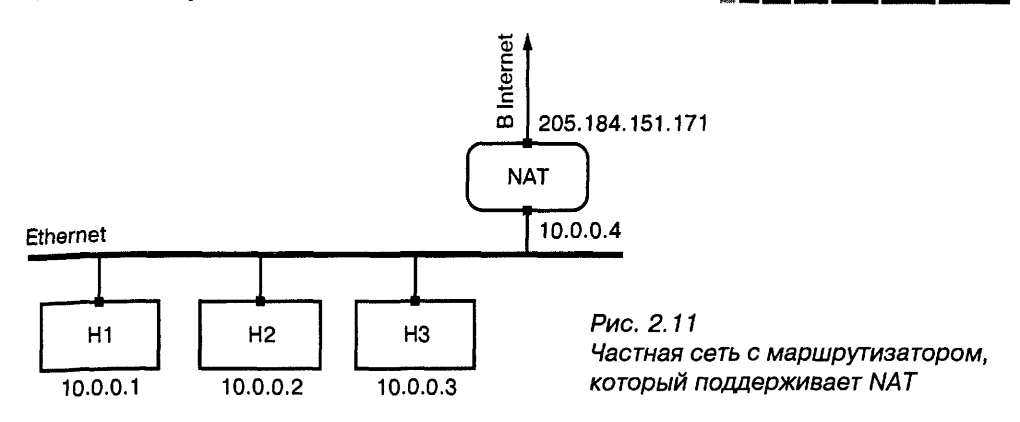


Рис. 2.11
Частная сеть с маршрутизатором, который поддерживает NAT

Поскольку показан только один глобальный адрес, ассоциированный с NAT, предположим, что маршрутизатор сконфигурирован с возможностью использования метода PAT. Статический режим и режим выбора из пула аналогичны методу PAT, но проще его, поскольку не нужно преобразовывать еще и номера портов.

Допустим, что хосту H2 надо отправить SYN-сегмент TCP по адресу 204.71.200.69 – на один из Web-серверов www.yahoo.com, – чтобы открыть соединение. На рис. 2.12а видно, что у сегмента, покидающего H2, адрес получателя равен 204.71.200.69.80, а адрес отправителя – 10.0.0.2.9600.

Примечание *Здесь использована стандартная нотация, согласно которой адрес, записанный в форме A.B.C.D.P означает IP-адрес A.B.C.D и порт P.*

В этом нет ничего особенного, за исключением того, что адрес отправителя принадлежит частной сети. Когда этот сегмент доходит до маршрутизатора, NAT должен заменить адрес отправителя на 205.184.151.171, чтобы Web-сервер на сайте Yahoo знал, куда посылать сегмент SYN/ACK и последующие. Поскольку во всех пакетах, исходящих от других хостов в частной сети, адрес отправителя также будет заменен на 205.184.151.171, NAT необходимо изменить еще и номер порта на некоторое уникальное значение, чтобы потом определять, какому хосту следует переправлять входящие пакеты. Исходящий порт 9600 преобразуется в 5555. Таким образом, у сегмента, доставленного на сайт Yahoo, адрес получателя будет 204.71.200.69.80, а адрес отправителя – 205.184.151.171.5555.

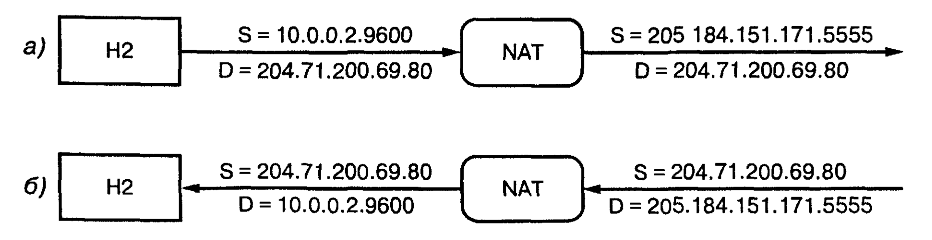


Рис. 2.12. Преобразование адресов портов

Из рис. 2.12б видно также, что в дошедшем до маршрутизатора ответе Yahoo адрес получателя равен 205.184.151.171.5555. NAT ищет этот номер порта в своей внутренней таблице и обнаруживает, что порт 5555 соответствует адресу 10.0.0.1.9600, так что после получения от маршрутизатора этого пакета в хосте H2 появится информация, что адрес отправителя равен 204.71.200.69.80, а адрес получателя – 10.0.0.1.9600.

Описанный здесь метод PAT выглядит довольно примитивно, но есть много усложняющих его деталей. Например, при изменении адреса отправителя или номера исходящего порта меняются как контрольная сумма заголовка IP-датаграммы, так и контрольная сумма TCP-сегмента, поэтому их необходимо скорректировать.

В качестве другого примера возможных осложнений рассмотрим протокол передачи файлов FTP (File Transfer Protocol) [Reynolds and Postel 1985]. Когда FTP-клиенту нужно отправить файл или принять его от FTP-сервера, серверу посылается команда PORT с указанием адреса и номера порта, по которому будет ожидаться соединение (для передачи данных) от сервера. При этом NAT нужно распознать TCP-сегмент, содержащий команду PORT протокола FTP, и подменить в ней адрес и порт. В команде PORT адрес и номер порта представлены в виде ASCII-строк, поэтому при их подмене может измениться размер сегмента. А это, в свою очередь, повлечет изменение порядковых номеров байтов. Так что NAT должен за этим следить, чтобы вовремя скорректировать порядковые номера в сегменте подтверждения ACK, а также в последующих сегментах с того же хоста.

Несмотря на все эти сложности, NAT работает неплохо и широко распространен. В частности, PAT – это естественный способ подключения небольших сетей к Internet в ситуации, когда имеется только одна точка выхода.

Резюме

В этом разделе показано, как схема NAT позволяет использовать один из блоков частных сетевых адресов для внутренних хостов, сохраняя при этом возможность выхода в Internet. Метод PAT, в частности, особенно полезен для небольших сетей, у которых есть только один глобально выделенный IP-адрес. К сожалению, поскольку PAT изменяет номер порта в исходящих пакетах, он может оказаться несовместимым с нестандартными протоколами, которые передают информацию о номерах портов в теле сообщения.

Совет 4. Разрабатывайте и применяйте каркасы приложений

Большинство приложений TCP/IP попадают в одну из четырех категорий:

- TCP-сервер;
- TCP-клиент;
- UDP-сервер;
- UDP-клиент.

В приложениях одной категории обычно встречается почти одинаковый «стартовый» код, который инициализирует все, что связано с сетью. Например, TCP-сервер должен поместить в поля структуры sockaddr_in адрес и порт получателя,

получить от системы сокет типа SOCK_STREAM, привязать к нему выбранный адрес и номер порта, установить опцию сокета SO_REUSEADDR (совет 23), вызвать listen, а затем быть готовым к приему соединения (или нескольких соединений) с помощью системного вызова accept.

На каждом из этих этапов следует проверять код возврата. А часть программы, занимающаяся преобразованием адресов, должна иметь дело как с числовыми, так и с символическими адресами и номерами портов. Таким образом, в любом TCP-сервере есть порядка 100 почти одинаковых строк кода для выполнения всех перечисленных выше задач. Один из способов решения этой проблемы – поместить стартовый код в одну или несколько библиотечных функций, которые приложение может вызвать. Эта стратегия использована в книге. Но иногда приложению нужна слегка видоизмененная последовательность инициализации. В таком случае придется либо написать ее с нуля, либо извлечь нужный фрагмент кода из библиотеки и подправить его.

Чтобы справиться и с такими ситуациями, можно построить каркас приложения, в котором уже есть весь необходимый код. Затем скопировать этот каркас, внести необходимые изменения, после чего заняться логикой самого приложения. Не имея каркаса, легко поддаться искушению и срезать некоторые углы, например, жестко «зашить» в приложение адреса (совет 29) или сделать еще что-то сомнительное. Разработав каркас, вы сможете убрать все типичные функции в библиотеку, а каркас оставить только для необычных задач.

Чтобы сделать программы переносимыми, следует определить несколько макросов, в которых скрыть различия между API систем UNIX и Windows. Например, в UNIX системный вызов для закрытия сокета называется close, а в Windows – closesocket. Версии этих макросов для UNIX показаны в листинге 2.1. Версии для Windows аналогичны, приведены в приложении 2. Доступ к этим макросам из каркасов осуществляется путем включения файла skel.h.

Листинг 2.1. Заголовочный файл skel.h

skel.h

```
#ifndef __SKEL_H__
#define __SKEL_H__

/* версия для UNIX */

#define INIT()              ( program_name = \
                            strrchr( argv[ 0 ], '/' ) ) ? \
                            program_name++ : \
                            ( program_name = argv[ 0 ] )
#define EXIT(s)             exit( s )
#define CLOSE(s)            if ( close( s ) ) error( 1, errno, \
                            "ошибка close " )
#define set_errno(e)        errno = ( e )
#define isvalidsock(s)      ( ( s ) >= 0 )

typedef int SOCKET;

#endif  /* __SKEL_H__ */
```

skel.h

Каркас TCP-сервера

Начнем с каркаса TCP-сервера. Затем можно приступить к созданию библиотеки, поместив в нее фрагменты кода из каркаса. В листинге 2.2 показана функция `main`.

Листинг 2.2. Функция main из каркаса tcpserver.skel

tcpserver.skel

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdarg.h>
#include <string.h>
#include <errno.h>
#include <netdb.h>
#include <fcntl.h>
#include <sys/time.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "skel.h"

char *program_name;

int main( int argc, char **argv )
   {
   struct sockaddr_in local;
   struct sockaddr_in peer;
   char *hname;
   char *sname;
   int peerlen;
   SOCKET s1;
   SOCKET s;
   const int on = 1;

   INIT();

   if ( argc == 2 )
   {
      hname = NULL;
      sname = argv[ 1 ];
   }
   else
   {
      hname = argv[ 1 ];
      sname = argv[ 2 ];
   }

   set_address( hname, sname, &local, "tcp" );
   s = socket( AF_INET, SOCK_STREAM, 0 );
   if ( !isvalidsock( s ) )
      error( 1, errno, "ошибка вызова socket" );
```

```
40    if ( setsockopt( s, SOL_SOCKET, SO_REUSEADDR, &on,
41         sizeof( on ) ) )
42       error( 1, errno, "ошибка вызова setsockopt" );
43    if ( bind( s, ( struct sockaddr * ) &local,
44         sizeof( local ) ) )
45       error( 1, errno, "ошибка вызова bind" );
46    if ( listen( s, NLISTEN ) )
47       error( 1, errno, "ошибка вызова listen" );
48    do
49    {
50       peerlen = sizeof( peer );
51       s1 = accept( s, ( struct sockaddr * )&peer, &peerlen );
52       if ( !isvalidsock( s1 ) )
53          error( 1, errno, "ошибка вызова accept" );
54       server( s1, &peer );
55       CLOSE( s1 );
56    } while ( 1 );
57    EXIT( 0 );
58 }
```

tcpserver.skel

Включаемые файлы и глобальные переменные

1-14 Включаем заголовочные файлы, содержащие объявления используемых стандартных функций.

25 Макрос `INIT` выполняет стандартную инициализацию, в частности, установку глобальной переменной `program_name` для функции `error` и вызов функции `WSAStartup` при работе на платформе Windows.

Функция main

26-35 Предполагается, что при вызове сервера ему будут переданы адрес и номер порта или только номер порта. Если адрес не указан, то привязываем к сокету псевдоадрес `INADDR_ANY`, разрешающий прием соединений по любому сетевому интерфейсу. В настоящем приложении в командной строке могут, конечно, быть и другие аргументы, обрабатывать их надо именно в этом месте.

36 Функция `set_address` записывает в поля переменной `local` типа `sockaddr_in` указанные адрес и номер порта. Функция `set_address` показана в листинге 2.3.

37-45 Получаем сокет, устанавливаем в нем опцию `SO_REUSEADDR` (совет 23) и привязываем к нему хранящиеся в переменной `local` адрес и номер порта.

46-47 Вызываем `listen`, чтобы сообщить ядру о готовности принимать соединения от клиентов.

48-56 Принимаем соединения и для каждого из них вызываем функцию `server`. Она может самостоятельно обслужить соединение или создать для этого новый процесс. В любом случае после возврата из функции

server соединение закрывается. Странная, на первый взгляд, конструкция do-while позволяет легко изменить код сервера так, чтобы он завершался после обслуживания первого соединения. Для этого достаточно вместо

```
while ( 1 );
```

написать

```
while ( 0 );
```

Далее обратимся к функции set_address. Она будет использована во всех каркасах. Это естественная кандидатура на помещение в библиотеку стандартных функций.

Листинг 2.3. Функция set_address

tcpserver.skel

```
static void set_address( char *hname, char *sname,
   struct sockaddr_in *sap, char *protocol )
{
   struct servent *sp;
   struct hostent *hp;
   char *endptr;
   short port;

   bzero( sap, sizeof( *sap ) );
   sap->sin_family = AF_INET;
   if ( hname != NULL )
   {
      if ( !inet_aton( hname, &sap->sin_addr ) )
      {
         hp = gethostbyname( hname );
         if ( hp == NULL )
            error( 1, 0, "неизвестный хост: %s\n", hname );
         sap->sin_addr = *( struct in_addr * )hp->h_addr;
      }
   }
   else
      sap->sin_addr.s_addr = htonl( INADDR_ANY );
   port = strtol( sname, &endptr, 0 );
   if ( *endptr == '\0' )
      sap->sin_port = htons( port );
   else
   {
      sp = getservbyname( sname, protocol );
      if ( sp == NULL )
         error( 1, 0, "неизвестный сервис: %s\n", sname );
      sap->sin_port = sp->s_port;
   }
}
```

tcpserver.skel

set_address

8-9 Обнулив структуру `sockaddr_in`, записываем в поле адресного семейства `AF_INET`.

10-19 Если *hname* не `NULL`, то предполагаем, что это числовой адрес в стандартной десятичной нотации. Преобразовываем его с помощью функции `inet_aton`, если `inet_aton` возвращает код ошибки, – пытаемся преобразовать *hname* в адрес с помощью `gethostbyname`. Если и это не получается, то печатаем диагностическое сообщение и завершаем программу.

20-21 Если вызывающая программа не указала ни имени, ни адреса хоста, устанавливаем адрес `INADDR_ANY`.

22-24 Преобразовываем *sname* в целое число. Если это удалось, то записываем номер порта в сетевом порядке (совет 28).

27-30 В противном случае предполагаем, что это символическое название сервиса и вызываем `getservbyname` для получения соответствующего номера порта. Если сервис неизвестен, печатаем диагностическое сообщение и завершаем программу. Заметьте, что `getservbyname` уже возвращает номер порта в сетевом порядке.

Поскольку иногда приходится вызывать функцию `set_address` напрямую, здесь приводится ее прототип:

```
#include "etcp.h"

void set_address( char *host, char *port,
   struct sockaddr_in *sap, char *protocol );
```

Последняя функция – `error` – показана в листинге 2.4. Это стандартная диагностическая процедура.

```
#include "etcp.h"

void error( int status, int err, char *format, ...);
```

Если *status* не равно 0, то `error` завершает программу после печати диагностического сообщения; в противном случае она возвращает управление. Если *err* не равно 0, то считается, что это значение системной переменной `errno`. При этом в конец сообщения дописывается соответствующая этому значению строка и числовое значение кода ошибки.

Далее в примерах постоянно используется функция `error`, поэтому добавим ее в библиотеку.

Листинг 2.4. Функция error

tcpserver.skel

```
void error( int status, int err, char *fmt, ... )
{
   va_list ap;
```

```
   va_start( ap, fmt );
   fprintf( stderr, "%s: ", program_name );
   vfprintf( stderr, fmt, ap );
   va_end( ap );
   if ( err )
      fprintf( stderr, ": %s (%d)\n", strerror( err ), err );
   if ( status )
      EXIT( status );
}
```

tcpserver.skel

В каркас включена также заглушка для функции `server`:

```
static void server(SOCKET s, struct sockaddr_in *peerp )
{
}
```

Каркас можно превратить в простое приложение, добавив код внутрь этой заглушки. Например, если скопировать файл `tcpserver.skel` в `hello.c` и заменить заглушку кодом

```
static void server(SOCKET s, struct sockaddr_in *peerp )
{
  send( s, "hello, world\n", 13, 0);
}
```

то получим сетевую версию известной программы на языке С. Если откомпилировать и запустить эту программу, а затем подсоединиться к ней с помощью программы telnet, то получится вполне ожидаемый результат:

```
bsd: $ hello 9000
[1] 1163
bsd: $ telnet localhost 9000
Trying 127.0.0.1...
Connected to localhost
Escape character '^]'.
hello, world
Connection closed by foreign host.
```

Поскольку каркас `tcpserver.skel` описывает типичную для TCP-сервера ситуацию, поместим большую часть кода `main` в библиотечную функцию `tcp_server`, показанную в листинге 2.5. Ее прототип выглядит следующим образом:

```
#include "etcp.h"

SOCKET tcp_server( char *host, char *port );
```

Возвращаемое значение: сокет в режиме прослушивания (в случае ошибки завершает программу).

Параметр *host* указывает на строку, которая содержит либо имя, либо IP-адрес хоста, а параметр *port* – на строку с символическим именем сервиса или номером порта, записанным в виде ASCII-строки.

Далее будем пользоваться функцией tcp_server, если не возникнет необходимость модифицировать каркас кода.

Листинг 2.5. Функция tcp_server

tcp_server.c
```
SOCKET tcp_server( char *hname, char *sname )
{
   struct sockaddr_in local;
   SOCKET s;
   const int on = 1;

   set_address( hname, sname, &local, "tcp" );
   s = socket( AF_INET, SOCK_STREAM, 0 );
   if ( !isvalidsock( s ) )
      error( 1, errno, "ошибка вызова socket" );

   if ( setsockopt( s, SOL_SOCKET, SO_REUSEADDR,
      ( char * )&on, sizeof( on ) ) )
      error( 1, errno, "ошибка вызова setsockopt" );

   if ( bind( s, ( struct sockaddr * ) &local,
       sizeof( local ) ) )
      error( 1, errno, "ошибка вызова bind" );

   if ( listen( s, NLISTEN ) )
      error( 1, errno, "ошибка вызова listen" );

   return s;
}
```
tcp_server.c

Каркас TCP-клиента

Рассмотрим каркас приложения TCP-клиента (листинг 2.6). Если не считать функции main и замены заглушки server заглушкой client, то код такой же, как для каркаса TCP-сервера.

Листинг 2.6. Функция main из каркаса tcpclient.skel

tcpclient.skel
```
int main( int argc, char **argv )
{
   struct sockaddr_in peer;
   SOCKET s;

   INIT();

   set_address( argv[ 1 ], argv[ 2 ], &peer, "tcp" );

   s = socket( AF_INET, SOCK_STREAM, 0 );
   if ( !isvalidsock( s ) )
         error( 1, errno, "ошибка вызова socket" );
   if ( connect( s, ( struct sockaddr * )&peer,
         sizeof( peer ) ) )
```

```
12         error( 1, errno, "ошибка вызова connect" );
13     client( s, &peer );
14     EXIT( 0 );
15 }
```

———————————————————————— *tcpclient.skel*

tcp_client.skel

6-9 Как и в случае `tcpserver.skel`, записываем в поля структуры `sockaddr_in` указанные адрес и номер порта, после чего получаем сокет.

10-11 Вызываем `connect` для установления соединения с сервером.

13 После успешного возврата из `connect` вызываем заглушку `client`, передавая ей соединенный сокет и структуру с адресом сервера.

Протестировать клиент можно, скопировав каркас в файл `helloc.c` и дописав в заглушку следующий код:

```
static void client( SOCKET s, struct sockaddr_in *peerp )
{
    int rc;
    char buf[ 120 ];

    for ( ;; )
    {
        rc = recv( s, buf, sizeof( buf ), 0 );
        if ( rc <= 0 )
            break;
        write( 1, buf, rc );
    }
}
```

Этот клиент читает из сокета данные и выводит их на стандартный вывод до тех пор, пока сервер не пошлет конец файла (EOF). Подсоединившись к серверу `hello`, получаете:

```
bsd: $ hello localhost 9000
hello, world
bsd: $
```

Поместим фрагменты кода `tcpclient.skel` в библиотеку, так же, как поступили с каркасом `tcpclient.skel`. Новая функция – `tcp_client`, приведенная в листинге 2.7, имеет следующий прототип:

```
#include "etcp.h"

SOCKET tcp_client( char *host, char *port );
```

Возвращаемое значение: соединенный сокет (в случае ошибки завершает программу).

Как и в случае `tcp_server`, параметр *`host`* содержит либо имя, либо IP-адрес хоста, а параметр *`port`* – символическое имя сервиса или номер порта в виде ASCII-строки.

Листинг 2.7. Функция tcp_client

tcp_client.c

```
SOCKET tcp_client( char *hname, char *sname )
{
   struct sockaddr_in peer;
   SOCKET s;

   set_address( hname, sname, &peer, "tcp" );
   s = socket( AF_INET, SOCK_STREAM, 0 );
   if ( !isvalidsock( s ) )
      error( 1, errno, "ошибка вызова socket" );

   if ( connect( s, ( struct sockaddr * )&peer,
        sizeof( peer ) ) )
      error( 1, errno, "ошибка вызова connect" );
   return s;
}
```

tcp_client.c

Каркас UDP-сервера

Каркас UDP-сервера в основном похож на каркас TCP-сервера. Его отличительная особенность – не нужно устанавливать опцию сокета SO_REUSEADDR и обращаться к системным вызовам accept и listen, поскольку UDL – это протокол, не требующий логического соединения (совет 1). Функция main из каркаса приведена в листинге 2.8.

Листинг 2.8. Функция main из каркаса udpserver.skel

udpserver.skel

```
int main( int argc, char **argv )
{
   struct sockaddr_in local;
   char *hname;
   char *sname;
   SOCKET s;

   INIT();

   if ( argc == 2 )
   {
      hname = NULL;
      sname = argv[ 1 ];
   }
   else
   {
      hname = argv[ 1 ];
      sname = argv[ 2 ];
   }

   set_address( hname, sname, &local, "udp" );
```

```
19    s = socket( AF_INET, SOCK_DGRAM, 0 );
20    if ( !isvalidsock( s ) )
21       error( 1, errno, "ошибка вызова socket" );
22    if ( bind( s, ( struct sockaddr * ) &local,
23        sizeof( local ) ) )
24       error( 1, errno, "ошибка вызова bind" );
25    server( s, &local );
26    EXIT( 0 );
27 }
```

udpserver.skel

udpserver.skel

18 Вызываем функцию set_address для записи в поля переменной local типа sockaddr_in адреса и номера порта, по которому сервер будет принимать датаграммы. Обратите внимание, что вместо "tcp" задается третьим параметром "udp".

19-24 Получаем сокет типа SOCK_DGRAM и привязываем к нему адрес и номер порта, хранящиеся в переменной local.

25 Вызываем заглушку server, которая будет ожидать входящие датаграммы.

Чтобы получить UDP-версию программы «hello world», следует скопировать каркас в файл udphelloc.c и вместо заглушки вставить следующий код:

```
static void server( SOCKET s, struct sockaddr_in *localp )
{
   struct sockaddr_in peer;
   int peerlen;
   char buf[ 1 ];

   for ( ;; )
   {
      peerlen = sizeof( peer );
      if ( recvfrom( s, buf, sizeof( buf ), 0,
         ( struct sockaddr * )&peer, &peerlen ) < 0 )
         error( 1, errno, "ошибка вызова recvfrom" );
      if ( sendto( s, "hello, world\n", 13, 0,
         ( struct sockaddr * )&peer, peerlen ) < 0 )
         error( 1, errno, "ошибка вызова sendto" );
   }
}
```

Прежде чем тестировать этот сервер, нужно разработать каркас UDP-клиента (листинг 2.10). Но сначала нужно вынести последнюю часть main в библиотечную функцию udp_server:

```
#include "etcp.h"

SOCKET udp_server( char *host, char *port );
```

Возвращаемое значение: UDP-сокет, привязанный к хосту *host* и порту *port* (в случае ошибки завершает программу).

Как обычно, параметры *host* и *port* указывают на строки, содержащие соответственно имя или IP-адрес хоста и имя сервиса либо номер порта в виде ASCII-строки.

Листинг 2.9. Функция udp_server

```
——————————————————————————————————————udp_server.c
SOCKET udp_server( char *hname, char *sname )
{
SOCKET s;
struct sockaddr_in local;

set_address( hname, sname, &local, "udp" );
s = socket( AF_INET, SOCK_DGRAM, 0 );
if ( !isvalidsock( s ) )
   error( 1, errno, "ошибка вызова socket" );
if ( bind( s, ( struct sockaddr * ) &local,
    sizeof( local ) ) )
   error( 1, errno, "ошибка вызова bind" );
return s;
}
——————————————————————————————————————udp_server.c
```

Каркас UDP-клиента

Функция main в каркасе UDP-клиента выполняет в основном запись в поля переменной peer указанных адреса и номера порта сервера и получает сокет типа SOCK_DGRAM. Она показана в листинге 2.10. Весь остальной код каркаса такой же, как для udpserver.skel.

Листинг 2.10. Функция main *из каркаса* udpclient.skel

```
——————————————————————————————————————udpclient.skel
int main( int argc, char **argv )
{
   struct sockaddr_in peer;
   SOCKET s;

   INIT();

   set_address( argv[ 1 ], argv[ 2 ], &peer, "udp" );
   s = socket( AF_INET, SOCK_DGRAM, 0 );
   if ( !isvalidsock( s ) )
      error( 1, errno, "ошибка вызова socket" );

   client( s, &peer );
   exit( 0 );
}
——————————————————————————————————————udpclient.skel
```

Теперь можно протестировать одновременно этот каркас и программу udphello, для чего необходимо скопировать udpclient.skel в файл udphelloc.c и вместо клиентской заглушки подставить такой код:

```
static void client( SOCKET s, struct sockaddr_in *peerp )
{
    int rc;
    int peerlen;
    char buf[ 120 ];

    peerlen = sizeof( *peerp );
    if ( sendto( s, "", 1, 0, ( struct sockaddr * )peerp,
        peerlen ) < 0 )
        error( 1, errno, "ошибка вызова sendto" );
    rc = recvfrom( s, buf, sizeof( buf ), 0,
        ( struct sockaddr * )peerp, &peerlen );
    if ( rc >= 0 )
        write( 1, buf, rc );
    else
        error( 1, errno, "ошибка вызова recvfrom" );

}
```

Функция `client` посылает серверу нулевой байт, читает возвращенную датаграмму, выводит ее в стандартное устройство вывода и завершает программу. Функции `recvfrom` в коде `udphello` вполне достаточно одного нулевого байта. После его приема она возвращает управление основной программе, которая и посылает ответную датаграмму.

При одновременном запуске обеих программ выводится обычное приветствие:

```
bsd: $ udphello 9000 &
[1] 448
bsd: $ updhelloc localhost 9000
hello, world
bsd: $
```

Как всегда, следует вынести стартовый код из `main` в библиотеку. Обратите внимание, что библиотечной функции, которой дано имя `udp_client` (листинг 2.11), передается третий аргумент – адрес структуры `sockaddr_in`; в нее будет помещен адрес и номер порта, переданные в двух первых аргументах.

```
#include "etcp.h"

SOCKET udp_client( char *host, char *port,
    struct sockaddr_in *sap );
```

Возвращаемое значение: UDP-сокет и заполненная структура `sockaddr_in` (в случае ошибки завершает программу).

Листинг 2.11. Функция udp_client

udp_client.c

```
SOCKET udp_client( char *hname, char *sname,
    struct sockaddr_in *sap )
{
    SOCKET s;
```

```
    set_address( hname, sname, sap, "udp" );
    s = socket( AF_INET, SOCK_DGRAM, 0 );
    if ( !isvalidsock( s ) )
       error( 1, errno, "ошибка вызова socket" );
    return s;
}
```
udp_client.c

Резюме

Прочитав данный раздел, вы узнали, как просто создать целый арсенал каркасов и библиотечных функций. Все построенные каркасы очень похожи и различаются только несколькими строками в стартовом коде внутри функции `main`. Таким образом, после написания первого каркаса пришлось лишь скопировать код и подправить эти несколько строк. Эта методика очень проста. Поэтому, чтобы создать несколько элементарных клиентов и серверов, потребовалось только вставить содержательный код вместо заглушек.

Использование каркасов и написание библиотечных функций закладывает тот фундамент, на котором далее легко строить приложения и небольшие тестовые программки для их проверки.

Совет 5. Предпочитайте интерфейс сокетов интерфейсу XTI/TLI

В мире UNIX в качестве интерфейса к коммуникационным протоколам, в частности к TCP/IP, в основном используются следующие два API:

- сокеты Беркли;
- транспортный интерфейс XTI(X/Open Transport Interface).

Интерфейс сокетов разработан в Университете г. Беркли штата Калифорния и вошел в состав созданной там же версии операционной системы UNIX. Он получил широкое распространение вместе с версией 4.2BSD (1983), затем был усовершенствован в версии 4.3BSD Reno (1990) и теперь включается практически во все версии UNIX. API сокетов присутствует и в других операционных системах. Так, Winsock API популярной в мире Microsoft Windows основан на сокетах из BSD [Winsock Group 1997].

API интерфейса XTI – это расширение *интерфейса к транспортному уровню* (Transport Layer Interface – TLI), который впервые появился в системе UNIX System V Release 3.0 (SVR3) компании AT&T. TLI задумывался как интерфейс, не зависящий от протокола, так как он сравнительно легко поддерживает новые протоколы. На его дизайн оказала значительное влияние модель протоколов OSI (совет 14). В то время многие полагали, что эти протоколы вскоре придут на смену TCP/IP. И поэтому, с точки зрения программиста TCP/IP, дизайн этого интерфейса далек от оптимального. Кроме того, хотя имена функций TLI очень похожи на используемые в API сокетов (только они начинаются с `t_`), их семантика в ряде случаев кардинально отличается.

Тот факт, что интерфейс TLI все еще популярен, возможно, объясняется его использованием с протоколами Internetwork Packet Exchange/Sequenced Packet Exchange (IPX/SPX) в системах фирмы Novell. Поэтому при переносе программ, написанных для IPX/SPX, под TCP/IP проще было воспользоваться тем же интерфейсом TLI [Kacker 1999].

В четвертой части первого тома книги «UNIX Network Programming» [Stevens 1998] имеется прекрасное введение в программирование XTI и подсистемы STREAMS. Представить, насколько отличается семантика XTI и сокетов, можно хотя бы по тому, что обсуждению XTI посвящено более 100 страниц.

Надеясь, что протоколы OSI все-таки заменят TCP/IP, многие производители UNIX-систем рекомендовали писать новые приложения с использованием TLI API. Одна фирма-производитель даже заявила, что интерфейс сокетов не будет поддерживаться в следующих версиях. Но такие прогнозы оказались несколько преждевременными.

Протоколы OSI можно считать устаревшими, но TLI и последовавший за ним XTI все еще поставляются в составе UNIX-систем, производных от System V. Поэтому при программировании для UNIX встает вопрос: что лучше использовать – сокеты или XTI?

Здесь необходимо напомнить, почему указанные протоколы называются интерфейсами. Для программиста TCP/IP это всего лишь разные способы доступа к стеку TCP/IP. Поскольку именно этот стек реализует коммуникационные протоколы, не имеет значения, какой API использует его клиент. Это означает, что приложение, написанное с помощью сокетов, может обмениваться данными с приложением на базе XTI. В системах типа SVR4 оба интерфейса обычно реализуются в виде библиотек, осуществляющих доступ к стеку TCP/IP с помощью подсистемы STREAMS.

Рассмотрим сначала интерфейс XTI. У него есть своя ниша в сетевом программировании. Поскольку он не зависит от протокола, с его помощью можно добавить в систему UNIX новый протокол, не имея доступа к коду ядра. Проектировщику протокола необходимо лишь реализовать транспортный провайдер в виде STREAMS-мультиплексора, связать его с ядром, а потом обращаться к нему через XTI.

Примечание *О том, как писать модули STREAMS, а также о программировании TLI и STREAMS вы можете прочесть в книге [Rago 1993].*

Обратите внимание, насколько специфична ситуация: нужно реализовать отсутствующий в системе протокол, когда нет доступа к исходным текстам ядра.

Примечание *Кроме того, этот протокол нужно разработать для системы SVR4 или любой другой, поддерживающей STREAMS и XTI/TLI. Начиная с версии Solaris 2.6, фирма Sun предоставляет такую же функциональность с помощью API сокетов.*

Иногда утверждают, что проще писать не зависящий от протокола код с помощью XTI/TLI [Rago 1993]. Конечно, «простота» – понятие субъективное, но

в разделе 11.9 книги «UNIX Network Programming» Стивенс с помощью сокетов реализовал простой, не зависящий от протокола сервер времени дня, который поддерживает IP версии 4, IP версии 6 и сокеты в адресном домене UNIX.

И, наконец, говорят, что при поддержке обоих интерфейсов сокеты обычно реализуются поверх TLI/XTI, так что TLI/XTI более эффективен. Это не так. Как отмечалось выше, в системах на базе SVR4 оба интерфейса обычно реализованы в виде библиотек, напрямую общающихся с подсистемой STREAMS. Фактически с версии Solaris 2.6 (Solaris – это версии SVR4, созданные фирмой Sun) сокеты реализованы непосредственно в ядре; обращение к ним происходит через вызовы системы.

Большое преимущество сокетов – переносимость. Поскольку сокеты есть практически во всех системах с XTI/TLI, их использование гарантирует максимальную переносимость. Даже если ваше приложение будет работать только под UNIX, так как большинство операционных систем, поддерживающих TCP/IP, предоставляет интерфейс сокетов. И лишь немногие системы, не принадлежащие к UNIX, содержат интерфейс XTI/TLI (если вообще такие существуют). Например, создание приложения, переносимого между UNIX и Microsoft Windows, – сравнительно несложная задача, так как Windows поддерживает спецификацию Winsock, в которой реализован API сокетов.

Еще одно преимущество сокетов в том, что этот интерфейс проще использовать, чем XTI/TLI. Поскольку XTI/TLI проектировался в основном как общий интерфейс (имеются в виду протоколы OSI), программисту приходится при его использовании писать больше кода, чем при работе с сокетами. Даже сторонники XTI/TLI согласны с тем, что для создания приложений TCP/IP следует предпочесть интерфейс сокетов.

Руководство «Введение в библиотеку подпрограмм», поставляемое в составе Solaris 2.6, дает такой совет по выбору API: «При всех обстоятельствах рекомендуется использовать API сокетов, а не XTI и TLI. Если требуется переносимость на другие системы, удовлетворяющие спецификации XPGV4v2, то следует использовать интерфейсы из библиотеки libxnet. Если же переносимость необязательна, то рекомендуется интерфейс сокетов из библиотек libsocket и libnsl, а не из libxnet. Если выбирать между XTI и TLI, то лучше пользоваться интерфейсом XTI (доступным через libxnet), а не TLI (доступным через libnsl)».

Резюме

В обычных ситуациях нет смысла использовать интерфейс XTI/TLI при программировании TCP/IP. Интерфейс сокетов проще и обладает большей переносимостью, а возможности обоих интерфейсов почти одинаковы.

Совет 6. Помните, что TCP – потоковый протокол

TCP – *потоковый протокол*. Это означает, что данные доставляются получателю в виде потока байтов, в котором нет понятий «сообщения» или «границы сообщения». В этом отношении чтение данных по протоколу TCP похоже на чтение из последовательного порта – заранее не известно, сколько байтов будет возвращено после обращения к функции чтения.

Представим, например, что имеется TCP-соединение между приложениями на хостах A и B. Приложение на хосте A посылает сообщения хосту B. Допустим, что у хоста A есть два сообщения, для отправки которых он дважды вызывает `send` – по разу для каждого сообщения. Естественно, эти сообщения передаются от хоста A к хосту B в виде раздельных блоков, каждое в своем пакете, как показано на рис. 2.13.

К сожалению, реальная передача данных вероятнее всего будет происходить не так. Приложение на хосте A вызывает `send`, и вроде бы данные сразу же передаются на хост B. На самом деле `send` обычно просто копирует данные в буфер стека TCP/IP на хосте A и тут же возвращает управление. TCP самостоятельно определяет, сколько данных нужно передать немедленно. В частности, он может вообще отложить передачу до более благоприятного момента. Принятие такого решения зависит от многих факторов, например: окна передачи (объем данных, которые хост B готов принять), окна перегрузки (оценка загруженности сети), максимального размера передаваемого блока вдоль пути (максимально допустимый объем данных для передачи в одном блоке на пути от A к B) и количества данных в выходной очереди соединения. Подробнее это рассматривается в совете 15. На рис. 2.14 показано только четыре возможных способа разбиения двух сообщений по пакетам. Здесь M_{11} и M_{12} – первая и вторая части сообщения M_1, а M_{21} и M_{22} – соответственно части M_2. Как видно из рисунка, TCP не всегда посылает все сообщения в одном пакете.

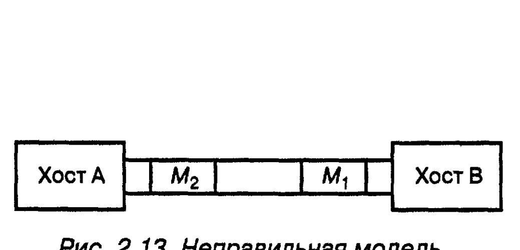


Рис. 2.13. Неправильная модель отправки двух сообщений

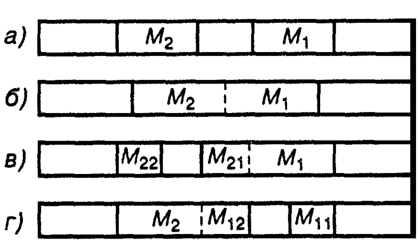


Рис. 2.14. Четыре возможных способа разбиения двух сообщений по пакетам

А теперь посмотрим на эту ситуацию с точки зрения приложения на хосте. В общем случае оно не имеет информации относительно количества возвращаемых TCP данных при обращении к системному вызову `recv`. Например, когда приложение на хосте B в первый раз читает данные, возможны следующие варианты

- данных еще нет, и приложение либо блокируется операционной системой либо `recv` возвращает индикатор отсутствия данных. Это зависит от того, был ли сокет помечен как блокирующий и какую семантику вызова `recv` поддерживает операционная система;
- приложение получает лишь часть данных из сообщения M_1, если TCP посланы пакеты так, как показано на рис. 2.14г;
- приложение получает все сообщение M_1, если TCP отправлены пакеты, как изображено на рис. 2.14а;

- приложение получает все сообщение M_1 и часть или все сообщение M_2, как представлено на рис. 2.14в.

Примечание *В действительности таких вариантов больше, так как здесь не учитывается возможность ошибки и получения конца файла. Кроме того, предполагается, что приложение читает все доступные данные.*

Значительную роль здесь играет время. Если приложением на хосте В первое сообщение прочитано не сразу после его отправки, а спустя некоторое время после того, как хост А послал второе, то будут прочитаны оба сообщения. Как видите, количество данных, доступных приложению в данный момент, – величина неопределенная.

Еще раз следует подчеркнуть, что TCP – потоковый протокол и, хотя данные передаются в IP-пакетах, размер пакета напрямую не связан с количеством данных, переданных TCP при вызове `send`. У принимающего приложения нет надежного способа определить, как именно данные распределены по пакетам, поскольку между соседними вызовами `recv` может прийти несколько пакетов.

Примечание *Это может произойти, даже если принимающее приложение реагирует очень быстро. Например, если один пакет потерян (вполне обычная ситуация в Internet, см. совет 12), а последующие пришли нормально, то TCP «придерживает» поступившие данные, пока не будет повторно передан и корректно принят пропавший пакет. В этот момент приложение получает данные из всех поступивших пакетов.*

TCP следит за количеством посланных и подтвержденных байтов, но не за их распределением по пакетам. В действительности, одни реализации при повторной передаче потерянного пакета посылают больше данных, другие – меньше. Все это настолько важно, что заслуживает выделения: *Для TCP-приложения нет понятия «пакет». Если приложение хоть как-то зависит от того, как TCP распределяет данные по пакетам, то его нужно перепроектировать.*

Так как количество возвращаемых в результате чтения данных непредсказуемо, вы должны быть готовы к обработке этой ситуации. Часто проблемы вообще не возникает. Допустим, вы пользуетесь для чтения данных стандартной библиотечной функцией `fgets`. При этом она сама будет разбивать поток байтов на строки (листинг 3.3). Иногда границы сообщений бывают важны, тогда приходится реализовывать их сохранение на прикладном уровне.

Самый простой случай – это сообщения фиксированной длины. Тогда вам нужно прочесть заранее известное число байтов из потока. В соответствии с вышесказанным, для этого *недостаточно* выполнить простое однократное чтение:

```
recv( s, msg, sizeof( msg ), 0 );
```

поскольку при этом можно получить меньше, чем `sizeof( msg )` байт (рис. 2.14г). Стандартный способ решения этой проблемы показан в листинге 2.12.

Листинг 2.12. Функция readn

readn.c

```
 1 int readn( SOCKET fd, char *bp, size_t len)
 2 {
 3    int cnt;
 4    int rc;
 5    cnt = len;
 6    while ( cnt > 0 )
 7    {
 8       rc = recv( fd, bp, cnt, 0 );
 9       if ( rc < 0 )                /* Ошибка чтения? */
10       {
11          if ( errno == EINTR )     /* Вызов прерван? */
12             continue;              /* Повторить чтение. */
13          return -1;                /* Вернуть код ошибки. */
14       }
15       if ( rc == 0 )               /* Конец файла? */
16          return len - cnt;         /* Вернуть неполный счетчик. */
17       bp += rc;
18       cnt -= rc;
19    }
20    return len;
21 }
```

readn.c

Функция `readn` используется точно так же, как `read`, только она не возвращает управления, пока не будет прочитано *len* байт или не получен конец файла или не возникнет ошибка. Ее прототип выглядит следующим образом:

```
#include «etcp.h»

int readn( SOCKET s, char *buf, size_t len );
```

Возвращаемое значение: число прочитанных байтов или –1 в случае ошибки.

Неудивительно, что `readn` использует ту же технику для чтения заданного числа байтов из последовального порта или иного потокового устройства, когда количество данных, доступных в данный момент времени, неизвестно. Обычно `readn` (с заменой типа `SOCKET` на `int` и `recv` на `read`) применяется во всех этих ситуациях.

Оператор `if`

```
if ( errno == EINTR )
   continue;
```

в строках 11 и 12 возобновляет выполнение вызова `recv`, если он прерван сигналом. Некоторые системы возобновляют прерванные системные вызовы автоматически, в таком случае эти две строки не нужны. С другой стороны, они не мешают, так что для обеспечения максимальной переносимости лучше их оставить.

Если приложение должно работать с сообщениями переменной длины, то в вашем распоряжении есть два метода. Во-первых, можно разделять записи специальными маркерами. Именно так надо поступить, используя стандартную функцию `fgets` для разбиения потока на строки. В этом случае естественным разделителем служит символ новой строки. Если маркер конца записи встретится в теле сообщения, то приложение-отправитель должно предварительно найти в сообщении все такие маркеры и экранировать их либо закодировать как-то еще, чтобы принимающее приложение не приняло их по ошибке за конец записи. Например, если в качестве признака конца записи используется символ-разделитель RS, то отправитель сначала должен найти все вхождения этого символа в тело сообщения и экранировать их, например, добавив перед каждым символ \. Это означает, что данные необходимо сдвинуть вправо, чтобы освободить место для символа экранирования. Его, разумеется, тоже необходимо экранировать. Так, если для экранирования используется символ \, то любое его вхождение в тело сообщения следует заменить на \\.

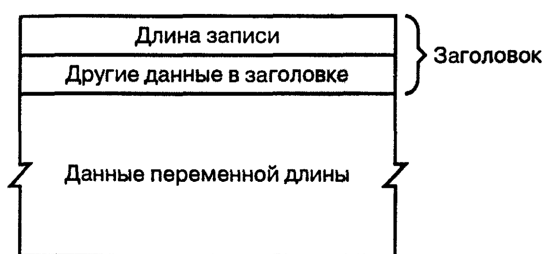


Рис. 2.15
Формат записи переменной длины

Принимающей стороне нужно просмотреть все сообщение, удалить символы экранирования и найти разделители записей. Поскольку при использовании маркеров конца записи все сообщение приходится просматривать дважды, этот метод лучше применять только при наличии «естественного» разделителя, например символа новой строки, разделяющего строки текста.

Другой метод работы с сообщениями переменной длины предусматривает снабжение каждого сообщения заголовком, содержащим (как минимум) длину следующего за ним тела. Этот метод показан на рис. 2.15.

Принимающее приложение читает сообщение в два приема: сначала заголовок фиксированной длины, и из него извлекается переменная длина тела сообщения, а затем – само тело. В листинге 2.13 приведен пример для простого случая, когда в заголовке хранится только длина записи.

Листинг 2.13. Функция для чтения записи переменной длины

readvrec.c

```
int readvrec( SOCKET fd, char *bp, size_t len )
{
	u_int32_t reclen;
	int rc;

	/* Прочитать длину записи. */
```

```
 6     rc = readn( fd, ( char * )&reclen, sizeof( u_int32_t ) );
 7     if ( rc != sizeof( u_int32_t ) )
 8        return rc < 0 ? -1 : 0;
 9     reclen = ntohl( reclen );
10     if ( reclen > len )
11     {
12        /*
13         *  Не хватает места в буфере для размещения данных -
14         *  отбросить их и вернуть код ошибки.
15         */

16        while ( reclen > 0 )
17        {
18           rc = readn( fd, bp, len );
19           if ( rc != len )
20              return rc < 0 ? -1 : 0;
21           reclen -= len;
22           if ( reclen < len )
23              len = reclen;
24        }
25        set_errno( EMSGSIZE );
26        return -1;
27     }

28     /* Прочитать саму запись */

29     rc = readn( fd, bp, reclen );
30     if ( rc != reclen )
31        return rc < 0 ? -1 : 0;
32     return rc;
33  }
```

readvrec.c

Чтение длины записи

6-8 Длина записи считывается в переменную `reclen`. Функция `readvrec` возвращает 0 (конец файла), если число байтов, прочитанных `readn`, не точно совпадает с размером целого, или –1 в случае ошибки.

9 Размер записи преобразуется из сетевого порядка в машинный. Подробнее об этом рассказывается в совете 28.

Проверка того, поместится ли запись в буфер

10-27 Проверяется, достаточна ли длина буфера, предоставленного вызывающей программой, для размещения в нем всей записи. Если места не хватит, то данные считываются в буфер частями по *len* байт, то есть, по сути, отбрасываются. Изъяв из потока отбрасываемые данные, функция присваивает переменной `errno` значение `EMSGSIZE` и возвращает –1.

Считывание записи

29-32 Наконец считывается сама запись. `readvrec` возвращает –1, 0 или `reclen` в зависимости от того, вернула ли `readn` код ошибки, неполный счетчик или нормальное значение.

Поскольку `readvrec` – функция полезная и ей найдется применение, необходимо записать ее прототип:

```
#include "etcp.h"

int readvrec( SOCKET s, char *buf, size_t len );
```

Возвращаемое значение: число прочитанных байтов или –1.

В листинге 2.14 дан пример простого сервера, который читает из TCP-соединения записи переменной длины с помощью `readvrec` и записывает их на стандартный вывод.

Листинг 2.14. vrs – сервер, демонстрирующие применение функции readvrec

vrs.c

```
 1 #include "etcp.h"

 2 int main( int argc, char **argv )
 3 {
 4    struct sockaddr_in peer;
 5    SOCKET s;
 6    SOCKET s1;
 7    int peerlen = sizeof( peer );
 8    int n;
 9    char buf[ 10 ];

10    INIT();
11    if ( argc == 2 )
12       s = tcp_server( NULL, argv[ 1 ] );
13    else
14       s = tcp_server( argv[ 1 ], argv[ 2 ] );
15    s1 = accept( s, ( struct sockaddr * )&peer, &peerlen );
16    if ( !isvalidsock( s1 ) )
17       error( 1, errno, "ошибка вызова accept" );
18    for ( ;; )
19    {
20       n = readvrec( s1, buf, sizeof( buf ) );
21       if ( n < 0 )
22          error( 0, errno, "readvrec вернула код ошибки" );
23       else if ( n == 0 )
24          error( 1, 0, "клиент отключился\n" );
25       else
26          write( 1, buf, n );
27    }
28    EXIT( 0 );       /* Сюда не попадаем. */
29 }
```

vrs.c

10-17 Инициализируем сервер и принимаем только одно соединение.

20-24 Вызываем `readvrec` для чтения очередной записи переменной длины. Если произошла ошибка, то печатается диагностическое сообщение

и читается следующая запись. Если readvrec возвращает EOF, то печатается сообщение и работа завершается.

26 Выводим записи на stdout.

В листинге 2.15 приведен соответствующий клиент, который читает сообщения из стандартного ввода, добавляет заголовок с длиной сообщения и посылает все это серверу.

Листинг 2.15. vrc – клиент, посылающий записи переменной длины

vrc.c

```
 1 #include "etcp.h"

 2 int main( int argc, char **argv )
 3 {
 4    SOCKET s;
 5    int n;
 6    struct
 7    {
 8       u_int32_t reclen;
 9    char buf[ 128 ];
10    } packet;

11    INIT();
12    s = tcp_client( argv[ 1 ], argv[ 2 ] );
13    while ( fgets( packet.buf, sizeof( packet.buf ), stdin )
14       != NULL )
15    {
16       n = strlen( packet.buf );
17       packet.reclen = htonl( n );
18       if ( send( s, ( char * )&packet,
19           n + sizeof( packet.reclen ), 0 ) < 0 )
20          error( 1, errno, "ошибка вызова send" );
21    }
22    EXIT( 0 );
23 }
```

vrc.c

Определение структуры packet

6-10 Определяем структуру packet, в которую будем помещать сообщение и его длину перед вызовом send. Тип данных u_int32_t – это беззнаковое 32-разрядное целое. Поскольку в Windows такого типа нет, в версии заголовочного файла skel.h для Windows приведено соответствующее определение типа.

Примечание *В этом примере есть одна потенциальная проблема, о которой следует знать. Предположим, что компилятор упаковывает данные в структуру, не добавляя никаких символов заполнения. Поскольку второй элемент – это массив байтов, в большинстве систем это предположение выполняется, но всегда нужно*

помнить о возможной недостоверности допущений о способе упаковки данных в структуру компилятором. Об этом будет рассказано в совете 24 при обсуждении способов для отправки нескольких элементов информации одновременно.

Connect, read и send

6-10 Клиент соединяется с сервером, вызывая функцию `tcp_client`.

13-21 Вызывается `fgets` для чтения строки из стандартного ввода. Эта строка помещается в пакет сообщения. С помощью функции `strlen` определяется длина строки. Полученное значение преобразуется в сетевой порядок байтов и помещается в поле `reclen` пакета. В конце вызывается `send` для отправки пакета серверу.

Другой способ отправки сообщений, состоящих из нескольких частей, рассматривается в совете 24.

Протестируем эти программы, запустив сервер на машине `sparc`, а клиент – на машине `bsd`. Поскольку результаты показаны рядом, видно, что поступает на вход клиенту и что печатает сервер. Чтобы сообщение строки 4 уместилось на странице, оно разбито на две строчки.

```
bsd: $ vrc sparc 8050  | sparc: $ vrs 8050
123                    | 123
123456789              | 123456789
1234567890             | vrs: readvrec вернула код ошибки:
                       |      Message too long (97)
12                     | 12
^C                     | vrs: клиент отключился
```

Поскольку длина буфера сервера равна 10 байт, функция `readvrec` возвращает код ошибки, когда отправляется 11 байт 1,..., 0,<LF>.

Резюме

Типичная ошибка, допускаемая начинающими сетевыми программистами, – в непонимании того, что TCP доставляет поток байтов, в котором нет понятия «границы записей». В TCP нет видимой пользователю концепции «пакета». Он просто передает поток байтов, и нельзя точно предсказать, сколько байтов будет возвращено при очередном чтении. В этом разделе рассмотрено несколько способов работы в таких условиях.

Совет 7. Не надо недооценивать производительность TCP

TCP – это сложный протокол, обеспечивающий базовую службу доставки IP-датаграмм надежностью и управлением потоком. В то же время UDP добавляет лишь контрольную сумму. Поэтому может показаться, что UDP должен быть на порядок быстрее TCP. Исходя из этого предположения, многие программисты полагают, что с помощью UDP можно достичь максимальной производительности.

Да, действительно, есть ситуации, когда UDP существенно быстрее TCP. Но иногда использование TCP оказывается эффективнее, чем применение UDP.

В сетевом программировании производительность любого протокола зависит от сети, приложения, нагрузки и других факторов, из которых не последнюю роль играет качество реализации. Единственный надежный способ узнать, какой протокол и алгоритм работают оптимально, – это протестировать их в предполагаемых условиях работы приложения. На практике это, конечно, не всегда осуществимо, но часто удается получить представление о производительности, воспользовавшись каркасами для построения простых программ, моделирующих ожидаемый сетевой трафик.

Перед созданием тестовых примеров необходимо разобраться, когда и почему производительность UDP больше, чем TCP. Прежде всего, поскольку TCP сложнее, он выполняет больше вычислений, чем UDP.

Примечание *В работе [Stevens, 1996] сообщается, что реализация TCP в системе 4.4 BSD содержит примерно 4500 строк кода на языке С в сравнении с 800 строками для UDP. Естественно, обычно выполняется намного меньше строк, но эти числа отражают сравнительную сложность кода.*

Но в типичной ситуации большая часть времени процессора в обоих протоколах тратится на копирование данных и вычисление контрольных сумм (совет 26), поэтому здесь нет большой разницы. В своей работе [Partridge 1993] Джекобсон описывает экспериментальную версию TCP, в которой для выполнения всего кода обычно требуется всего 30 машинных инструкций RISC (исключая вычисление контрольных сумм и копирование данных в буфер пользовательской программы, которые производятся одновременно).

Нужно отметить, что для обеспечения надежности TCP должен посылать подтверждения (АСК-сегменты) на каждый принятый пакет. Это несколько увеличивает объем обработки на обоих концах. Во-первых, принимающая сторона может включить АСК в состав данных, которые она посылает отправителю. В действительности во многих реализациях TCP отправка АСК задерживается на несколько миллисекунд: предполагается, что приложение-получатель вскоре сгенерирует ответ на пришедший сегмент. Во-вторых, TCP не обязан подтверждать каждый сегмент. В большинстве реализаций при нормальных условиях АСК посылается только на каждый второй сегмент.

Примечание *RFC 1122 [Braden 1989] рекомендует откладывать посылку АСК до 0,5 с при подтверждении каждого второго сегмента.*

Еще одно принципиальное отличие между TCP и UDP в том, что TCP требует логического соединения (совет 1) и, значит, необходимо заботиться об его установлении и разрыве. Для установления соединения обычно требуется обменяться тремя сегментами. Для разрыва соединения нужно четыре сегмента, которые, кроме последнего, часто можно скомбинировать с сегментами, содержащими данные.

Предположим, что время, необходимое для разрыва соединения в большинстве случаев не расходуется зря, поскольку одновременно передаются данные. Следует выяснить, что же происходит во время установления соединения. Как показано на рис. 2.16, клиент начинает процедуру установления соединения, посылая серверу сегмент SYN (синхронизация). В этом сегменте указывается порядковый номер, который клиент присвоит первому посланному байту, а также другие параметры соединения. В частности, *максимальный размер сегмента* (MSS), который клиент готов принять, и начальный размер окна приема. Сервер в ответ посылает свой сегмент SYN, который также содержит подтверждение ACK на сегмент SYN клиента. И, наконец, клиент отсылает ACK на сегмент SYN сервера. На этом процедура установления соединения завершается. Теперь клиент может послать свой первый сегмент данных.

На рис. 2.16 RTT (round-trip time) – это период кругового обращения, то есть время, необходимое пакету для прохождения с одного хоста на другой и обратно. Для установления соединения нужно полтора таких периода.

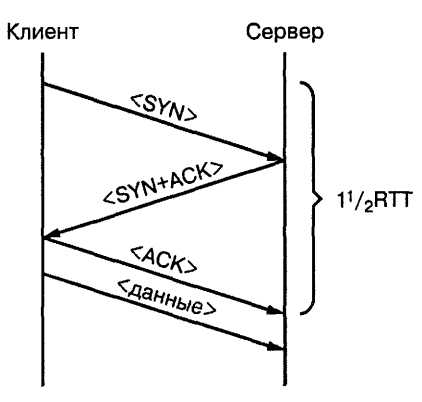


Рис. 2.16. Установление соединения

При длительном соединении между клиентом и сервером (например, клиент и сервер обмениваются большим объемом данных) указанный период «размазывается» между всеми передачами данных, так что существенного влияния на производительность это не оказывает. Однако если речь идет о простой транзакции, в течение которой клиент посылает запрос, получает ответ и разрывает соединение, то время инициализации составляет заметную часть от времени всей транзакции. Таким образом, следует ожидать, что UDP намного превосходит TCP по производительности именно тогда, когда приложение организует короткие сеансы связи. И, наоборот, TCP работает быстрее, когда соединение поддерживается в течении длительного времени при передаче больших объемов данных.

Чтобы протестировать сравнительную производительность TCP и UDP, а заодно продемонстрировать, как пишутся небольшие тестовые программки, создадим несколько простых серверов и клиентов. Здесь имеется в виду не полнофункциональная контрольная задача, а лишь определение производительности двух протоколов при передаче большого объема данных. Примером такого рода приложения служит протокол FTP.

Источник и приемник на базе UDP

В случае UDP клиент посылает нефиксированное количество датаграмм, которые сервер читает, подсчитывает и отбрасывает. Исходный текст клиента приведен в листинге 2.16.

Листинг 2.16. UDP-клиент, посылающий произвольное число датаграмм

udpsource.c

```
1 #include "etcp.h"
2 int main( int argc, char **argv )
3 {
4    struct sockaddr_in peer;
5    SOCKET s;
6    int rc;
7    int datagrams;
8    int dgramsz = 1440;
9    char buf[ 1440 ];

10   INIT();
11   datagrams = atoi( argv[ 2 ] );
12   if ( argc > 3 )
13      dgramsz = atoi( argv[ 3 ] );
14   s = udp_client( argv[ 1 ], "9000", &peer );
15   while ( datagrams-- > 0 )
16   {
17      rc = sendto( s, buf, dgramsz, 0,
18         ( struct sockaddr * )&peer, sizeof( peer ) );
19      if ( rc <= 0 )
20         error( 0, errno, "ошибка вызова sendto" );
21   }
22   sendto( s, "", 0, 0,
23      ( struct sockaddr * )&peer, sizeof( peer ) );
24   EXIT( 0 );
25 }
```

udpsource.c

10-14 Читаем из командной строки количество посылаемых датаграмм и их размер (второй параметр необязателен). Подготавливаем в переменной peer UDP-сокет с адресом сервера. Вопреки совету 29 номер порта 9000 жестко «зашит» в код.

15-21 Посылаем указанное количество датаграмм серверу.

22-23 Посылаем серверу последнюю датаграмму, содержащую нулевой байт. Для сервера она выполняет роль конца файла.

Текст сервера в листинге 2.17 еще проще.

Листинг 2.17. Приемник датаграмм

udpsink.c

```
#include "etcp.h"
int main( int argc, char **argv )
{
   SOCKET s;
   int rc;
   int datagrams = 0;
   int rcvbufsz = 5000 * 1440;
   char buf[ 1440 ];
```

```
 9     INIT();
10     s = udp_server( NULL, "9000" );
11     setsockopt( s, SOL_SOCKET, SO_RCVBUF,
12        ( char * )&rcvbufsz, sizeof( int ) );
13     for( ;; )
14     {
15        rc = recv( s, buf, sizeof( buf ), 0 );
16        if ( rc <= 0 )
17           break;
18        datagrams++;
19     }
20     error( 0, 0, "получено датаграмм: %d \n", datagrams );
21     EXIT( 0 );
22 }
```

udpsink.c

10 Подготавливаем сервер к приему датаграмм из порта 9000 с любого интерфейса.

11-12 Выделяем память для буфера на 5000 датаграмм длиной до 1440 байт.

Примечание *Здесь устанавливается размер буфера 7200000 байт, но нет гарантии, что операционная система выделит столько памяти. Хост, работающий под управлением системы BSD, выделил буфер размером 41600 байт. Этим объясняется потеря датаграмм, которая будет рассмотрена далее.*

13-19 Читаем и подсчитываем датаграммы, пока не придет пустая датаграмма или не произойдет ошибка.

20 Выводим число полученных датаграмм на `stdrerr`.

Источник и приемник на базе TCP

В совете 32 объясняется, что повысить производительность TCP можно за счет выбора правильного размера буферов передачи и приема. Нужно установить размер буфера приема для сокета сервера и размер буфера передачи для сокета клиента.

Поскольку в функциях `tcp_server` и `tcp_client` используются размеры буферов по умолчанию, следует воспользоваться не библиотекой, а каркасами из совета 4. Сообщать TCP размеры буферов нужно во время инициализации соединения, то есть до вызова `listen` в сервере и до вызова `connect` в клиенте. Поэтому невозможно воспользоваться функциями `tcp_server` и `tcp_client`, так как к моменту возврата из них обращение к `listen` или `connect` уже произошло. Начнем с клиента, его код приведен в листинге 2.18.

Листинг 2.18. Функция main TCP-клиента, играющего роль источника

tcpsource.c

```
int main( int argc, char **argv )
{
```

```
    struct sockaddr_in peer;
    char *buf;
    SOCKET s;
    int c;
    int blks = 5000;
    int sndbufsz = 32 * 1024;
    int sndsz = 1440;  /* MSS для Ethernet по умолчанию. */

    INIT();
    opterr = 0;
    while ( ( c = getopt( argc, argv, "s:b:c:" ) ) != EOF )
    {
        switch ( c )
        {
            case "s" :
                sndsz = atoi( optarg );
                break;

            case "b" :
                sndbufsz = atoi( optarg );
                break;

            case "c" :
                blks = atoi( optarg );
                break;

            case "?" :
                error( 1, 0, "некорректный параметр: %c\n", c );
        }
    }
    if ( argc <= optind )
        error( 1, 0, "не задано имя хоста\n" );

    if ( ( buf = malloc( sndsz ) ) == NULL )
        error( 1, 0, "ошибка вызова malloc\n" );
    set_address( argv[ optind ], "9000", &peer, "tcp" );
    s = socket( AF_INET, SOCK_STREAM, 0 );
    if ( !isvalidsock( s ) )
        error( 1, errno, "ошибка вызова socket" );

    if ( setsockopt( s, SOL_SOCKET, SO_SNDBUF,
        ( char * )&sndbufsz, sizeof( sndbufsz ) ) )
        error( 1, errno, "ошибка вызова setsockopt с опцией
        SO_SNDBUF" );

    if ( connect( s, ( struct sockaddr * )&peer,
         sizeof( peer ) ) )
        error( 1, errno, "ошибка вызова connect" );

    while( blks-- > 0 )
        send( s, buf, sndsz, 0 );
    EXIT( 0 );
}
```

tcpsource.c

main

12-30 В цикле вызываем `getopt` для получения и обработки параметров из командной строки. Поскольку эта программа будет использоваться и далее, то делаем ее конфигурируемой в большей степени, чем необходимо для данной задачи. С помощью параметров в командной строке можно задать размер буфера передачи сокета, количество данных, передаваемых при каждой операции записи в сокет, и число операций записи.

31-42 Это стандартный код инициализации TCP-клиента, только добавлено еще обращение к `setsockopt` для установки размера буфера передачи, а также с помощью функции `malloc` выделен буфер запрошенного размера для размещения данных, посылаемых при каждой операции записи. Обратите внимание, что инициализировать память, на которую указывает `buf`, не надо, так как в данном случае безразлично, какие данные посылать.

43-44 Вызываем функцию `send` нужное число раз.

Функция `main` сервера, показанная в листинге 2.19, взята из стандартного каркаса с добавлением обращения к функции `getopt` для получения из командной строки параметра, задающего размер буфера приема сокета, а также вызов функции `getsockopt` для установки размера буфера.

Листинг 2.19. Функция main TCP-сервера, играющего роль приемника

tcpsink.c

```
int main( int argc, char **argv )
{
   struct sockaddr_in local;
   struct sockaddr_in peer;
   int peerlen;
   SOCKET s1;
   SOCKET s;
   int c;
   int rcvbufsz = 32 * 1024;
   const int on = 1;

   INIT();
   opterr = 0;
   while ( ( c = getopt( argc, argv, "b:" ) ) != EOF )
   {
      switch ( c )
      {

         case "b" :
            rcvbufsz = atoi( optarg );
            break;

         case "?" :
            error( 1, 0, "недопустимая опция: %c\n", c );
      }
   }
```

```
    set_address( NULL, "9000", &local, "tcp" );
    s = socket( AF_INET, SOCK_STREAM, 0 );
    if ( !isvalidsock( s ) )
        error( 1, errno, "ошибка вызова socket" );

    if ( setsockopt( s, SOL_SOCKET, SO_REUSEADDR,
        ( char * )&on, sizeof( on ) ) )
        error( 1, errno, "ошибка вызова setsockopt SO_REUSEADDR" );

    if ( setsockopt( s, SOL_SOCKET, SO_RCVBUF,
        ( char * )&rcvbufsz, sizeof( rcvbufsz ) ) )
        error( 1, errno, "ошибка вызова setsockopt SO_RCVBUF" );

    if ( bind( s, ( struct sockaddr * ) &local,
         sizeof( local ) ) )
        error( 1, errno, "ошибка вызова bind" );

    listen( s, 5 );
    do
    {
        peerlen = sizeof( peer );
        s1 = accept( s, ( struct sockaddr * )&peer, &peerlen );
        if ( !isvalidsock( s1 ) )
            error( 1, errno, "ошибка вызова accept" );
        server( s1, rcvbufsz );
        CLOSE( s1 );
    } while ( 0 );
    EXIT( 0 );
}
```

tcpsink.c

Функция server читает и подсчитывает поступающие байты, пока не обнаружит конец файла (совет 16) или не возникнет ошибка. Она выделяет память под буфер того же размера, что и буфер приема сокета, чтобы прочитать максимальное количество данных за одно обращение к recv. Текст функции server приведен в листинге 2.20.

Листинг 2.20. Функция server

tcpsink.c

```
static void server( SOCKET s, int rcvbufsz )
{
    char *buf;
    int rc;
    int bytes = 0;

    if ( ( buf = malloc( rcvbufsz ) ) == NULL )
        error( 1, 0, "ошибка вызова malloc\n" );
    for ( ;; )
    {
        rc = recv( s, buf, rcvbufsz, 0 );
        if ( rc <= 0 )
            break;
```

```
    bytes += rc;
}
error( 0, 0, "получено байт: %d\n", bytes );
}
```

tcpsink.c

Для измерения сравнительной производительности протоколов TCP и UDP при передаче больших объемов данных запустим клиента на машине `bsd`, а сервер – на `localhost`. Физически хосты `bsd` и `localhost` – это, конечно, одно и то же, но, как вы увидите, результаты работы программы в значительной степени зависят от того, какое из этих имен использовано. Сначала запустим клиента и сервер на одной машине, чтобы оценить производительность TCP и UDP, устранив влияние сети. В обоих случаях сегменты TCP или датаграммы UDP инкапсулируются в IP-датаграммах и посылаются возвратному интерфейсу `lo0`, который немедленно переправляет их процедуре обработки IP-входа, как показано на рис. 2.17.

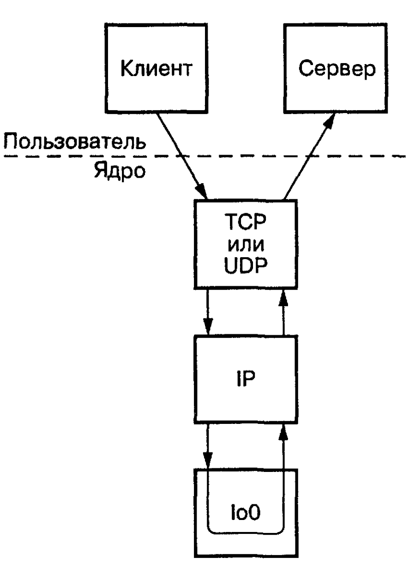


Рис. 2.17. Возвратный интерфейс

Каждый тест был выполнен 50 раз с заданным размером датаграмм (в случае UDP) или числом передаваемых за один раз байтов (в случае TCP), равным 1440. Эта величина выбрана потому, что она близка к максимальному размеру сегмента, который TCP может передать по локальной сети на базе Ethernet.

Примечание *Это число получается так. В одном фрейме Ethernet может быть передано не более 1500 байт. Каждый заголовок IP и TCP занимает 20 байт, так что остается 1460. Еще 20 байт резервировано для опций TCP. В системе BSD TCP посылает 12 байт с опциями, поэтому в этом случае максимальный размер сегмента составляет 1448 байт.*

В табл. 2.2 приведены результаты, усредненные по 50 прогонам. Для каждого протокола указано три времени: по часам – время с момента запуска до завершения работы клиента; пользовательское – проведенное программой в режиме пользователя; системное – проведенное программой в режиме ядра. В колонке «Мб/с» указан результат деления общего числа посланных байтов на время по часам. В колонке «Потеряно» для UDP приведено среднее число потерянных датаграмм.

Первое, что бросается в глаза, – TCP работает намного быстрее, когда в качестве имени сервера выбрано `localhost`, а не `bsd`. Для UDP это не так – заметной разницы в производительности нет. Чтобы понять, почему производительность TCP так возрастает, когда клиент отправляет данные хосту `localhost`, запустим

программу netstat (совет 38) с опцией -i. Здесь надо обратить внимание на две строки (ненужная информация опущена):

```
Name  Mtu   Network  Address
ed0   1500  172.30   bsd
lo0   16384 127      localhost
```

Таблица 2.2. Сравнение производительности TCP и UDP при количестве посылаемых байтов, равном 1440

TCP				
Сервер	**Время по часам**	**Пользовательское время**	**Системное время**	**Мб/с**
bsd	2,88	0,0292	1,4198	2,5
localhost	0,9558	0,0096	0,6316	7,53
sparc	7,1882	0,016	16226	1,002

UDP					
Сервер	**Время по часам**	**Пользовательское время**	**Системное время**	**Мб/с**	**Потеряно**
bsd	1,9618	0,0316	1,1934	3,67	336
localhost	1,9748	0,031	1,1906	3,646	272
sparc	5,8284	0,0564	0,844	1,235	440

Как видите, максимальный размер передаваемого блока (MTU – maximum transmission unit) для bsd равен 1500, а для localhost – 16384.

Примечание *Такое поведение свойственно реализациям TCP в системах, производных от BSD. Например, в системе Solaris это уже не так. При первом построении маршрута к хосту* bsd *в коде маршрутизации предполагается, что хост находится в локальной сети, поскольку сетевая часть IP-адреса совпадает с адресом интерфейса Ethernet. И лишь при первом использовании маршрута TCP обнаруживает, что он ведет на тот же хост и переключается на возвратный интерфейс. Однако к этому моменту все метрики маршрута, в том числе и MTU, уже установлены в соответствии с интерфейсом к локальной сети.*

Это означает, что при посылке данных на localhost TCP может отправлять сегменты длиной до 16384 байт (или 16384 – 20 – 20 – 12 = 16332 байт). Однако при посылке данных на хост bsd число байт в сегменте не превышает 1448 (как было сказано выше). Но чем больше размер сегментов, тем меньшее их количество приходится посылать, а это значит, что требуется меньший объем обработки, и соответственно снижаются накладные расходы на добавление к каждому сегменту заголовков IP и TCP. А результат налицо – обмен данными с хостом localhost происходит в три раза быстрее, чем с хостом bsd.

Можно заметить, что на хосте `localhost` TCP работает примерно в два раза быстрее, чем UDP. Это также связано с тем, что TCP способен объединять несколько блоков по 1440 байт в один сегмент, тогда как UDP посылает отдельно каждую датаграмму длиной 1440 байт.

Следует отметить, что в локальной сети UDP примерно на 20% быстрее TCP, но потеря датаграмм значительнее. Потери имеют место даже тогда, когда и сервер, и клиент работают на одной машине; связаны они с исчерпанием буферов. Хотя передача 5000 датаграмм на максимально возможной скорости – это скорее отклонение, чем нормальный режим работы, но все же следует иметь в виду возможность такого результата. Это означает, что UDP не дает *никакой* гарантии относительно доставки данной датаграммы, даже если оба приложения работают на одной машине.

По результатам сравнения сеансов с хостами `localhost` и `bsd` можно предположить, что на производительность влияет также длина посылаемых датаграмм. Например, если прогнать те же тесты с блоком длиной 300 байт, то, как следует из табл. 2.3, TCP работает быстрее UDP и на одной машине, и в локальной сети.

Из этих примеров следует важный вывод: нельзя строить априорные предположения о сравнительной производительности TCP и UDP. При изменении условий, даже очень незначительном, показатели производительности могут очень резко измениться. Для обоснованного выбора протокола лучше сравнить их производительность на контрольной задаче (совет 8). Когда это неосуществимо на практике, все же можно написать небольшие тестовые программы для получения хотя бы приблизительного представления о том, чего можно ожидать.

Таблица. 2.3. Сравнение производительности TCP и UDP при количестве посылаемых байтов, равном 300

TCP				
Сервер	**Время по часам**	**Пользовательское время**	**Системное время**	**Мб/с**
bsd	1,059	0,0124	0,445	1,416
sparc	1,5552	0,0084	1,2442	0,965

UDP					
Сервер	**Время по часам**	**Пользовательское время**	**Системное время**	**Мб/с**	**Потеряно**
bsd	1,6324	0,0324	0,9998	0,919	212
sparc	1,9118	0,0278	1,4352	0,785	306

Если говорить о практической стороне вопроса, то современные реализации TCP достаточно эффективны. Реально продемонстрировано, что TCP может работать со скоростью аппаратуры на стомегабитных сетях FDDI. В недавних экспериментах были достигнуты почти гигабитные скорости при работе на персональном компьютере [Gallatin et al. 1999].

Примечание *29 июля 1999 года исследователи из Университета Дьюка на рабочей станции XP1000 производства DEC/Compaq на базе процессора Alpha в сети Myrinet получили скорости передачи порядка гигабита в секунду. В экспериментах использовался стандартный стек TCP/IP из системы FreeBSD 4.0, модифицированный по технологии сокетов без копирования (zero-copy sockets). В том же эксперименте была получена скорость более 800 Мбит/с на персональном компьютере PII 450 МГц и более ранней версии сети Myrinet. Подробности можно прочитать на Web-странице http://www.cs.duke.edu/ari/trapeze.*

Резюме

UDP не всегда быстрее, чем TCP. На сравнительную производительность обоих протоколов влияют разные факторы, и для каждого конкретного случая желательно проверять быстродействие на контрольных задачах.

Совет 8. Не надо заново изобретать TCP

Как сказано в совете 7, UDP может быть намного производительнее TCP в простых приложениях, где есть один запрос и один ответ. Это наводит на мысль использовать в транзакционных задачах такого рода именно UDP. Однако протокол UDP не слишком надежен, поэтому эта обязанность лежит на приложении.

Как минимум, это означает, что приложение должно позаботиться о тех датаграммах, которые теряются или искажаются при передаче. Многие начинающие сетевые программисты полагают, что при работе в локальной сети такая возможность маловероятна, и потому полностью игнорируют ее. Но в совете 7 было показано, как легко можно потерять датаграммы даже тогда, когда клиент и север находятся на одной машине. Поэтому не следует забывать о защите от потери датаграмм.

Если же приложение будет работать в глобальной сети, то также возможен приход датаграмм не по порядку. Это происходит, когда между отправителем и получателем было несколько маршрутов.

В свете вышесказанного можно утверждать, что любое достаточно устойчивое UDP-приложение должно обеспечивать:

- повторную посылку запроса, если ответ не поступил в течение разумного промежутка времени;
- проверку соответствия между ответами и запросами.

Первое требование можно удовлетворить, если при посылке каждого запроса взводить таймер, называемый *таймером ретрансмиссии* (retransmission timer), или RTO-таймером. Если таймер срабатывает до получения ответа, то запрос посылается повторно. В совете 20 будет рассмотрено несколько способов эффективного решения этой задачи. Второе требование легко реализуется, если в каждый запрос включить его порядковый номер и обеспечить возврат этого номера сервером вместе с ответом.

Если приложение будет работать в Internet, то фиксированное время срабатывания RTO-таймера не годится, поскольку период кругового обращения (RTT) между двумя хостами может существенно меняться даже за короткий промежуток времени. Поэтому хотелось бы корректировать значение RTO-таймера в зависимости от условий в сети. Кроме того, если RTO-таймер срабатывает, следует увеличить его продолжительность перед повторной передачей, поскольку она, скорее всего, была слишком мала. Это требует некоторой экспоненциальной корректировки (exponential backoff) RTO при повторных передачах.

Далее, если приложение реализует что-либо большее, чем простой протокол запрос-ответ, когда клиент посылает запрос и ждет ответа, не посылая дополнительных датаграмм, или ответ сервера состоит из нескольких датаграмм, то необходим какой-то механизм управления потоком. Например, если сервер – это приложение, работающее с базой данных о кадрах, а клиент просит послать имена и адреса всех сотрудников конструкторского отдела, то ответ будет состоять из нескольких записей, каждая из которых посылается в виде отдельной датаграммы. Если управление потоком отсутствует, то при этом может быть исчерпан пул буферов клиента. Обычный способ решения этой проблемы – скользящее окно типа того, что используется в TCP (только подсчитываются не байты, а датаграммы).

И, наконец, если приложение передает подряд несколько датаграмм, необходимо позаботиться об управлении перегрузкой. В противном случае такое приложение может легко стать причиной деградации пропускной способности, которая затронет всех пользователей сети.

Все перечисленные действия, которые должно предпринять основанное на протоколе UDP приложение для обеспечения надежности, – это, по сути, вариант TCP. Иногда на это приходится идти. Ведь существуют транзакционные приложения, в которых накладные расходы, связанные с установлением и разрывом TCP-соединения, близки или даже превышают затраты на передачу данных.

Примечание *Обычный пример – это система доменных имен (Domain Name System – DNS), которая используется для отображения доменного имени хоста на его IP-адрес. Когда вводится имя хоста* `www.rfc-editor.org` *в Web-браузере, реализованный внутри браузера клиент DNS посылает DNS-серверу UDP-датаграмму с запросом IP-адреса, ассоциированного с этим именем. Сервер в ответ посылает датаграмму, содержащую IP-адрес 128.9.160.27. Подробнее система DNS обсуждается в совете 29.*

Тем не менее необходимо тщательно изучить природу приложения, чтобы понять, стоит ли заново реализовывать TCP. Если приложению требуется надежность TCP, то, быть может, правильное решение – это использование TCP.

Маловероятно, что функциональность TCP, продублированная на прикладном уровне, будет реализована столь же эффективно, как в настоящем TCP. Отчасти это связано с тем, что реализации TCP – это плод многочисленных экспериментов и научных исследований. С годами TCP эволюционировал по мере того,

как публиковались наблюдения за его работой в различных условиях и сетях, в том числе и Internet.

Кроме того, TCP почти всегда исполняется в контексте ядра. Чтобы понять, почему это может повлиять на производительность, представьте себе, что происходит при срабатывании RTO-таймера в вашем приложении. Сначала ядру нужно «пробудить» приложение, для чего необходимо контекстное переключение из режима ядра в режим пользователя. Затем приложение должно послать данные. Для этого требуется еще одно контекстное переключение (на этот раз в режим ядра), в ходе которого данные из датаграммы копируются в буферы ядра. Ядро выбирает маршрут следования датаграммы, передает ее подходящему сетевому интерфейсу и возвращает управление приложению – снова контекстное переключение. Приложение должно заново взвести RTO-таймер, для чего приходится вновь переключаться.

А теперь обсудим, что происходит при срабатывании RTO-таймера внутри TCP. У ядра уже есть сохраненная копия данных, нет необходимости повторно копировать их из пространства пользователя. Не нужны и контекстные переключения. TCP заново посылает данные. Основная работа связана с передачей данных из буферов ядра сетевому интерфейсу. Повторные вычисления не требуются, так как TCP сохранил всю информацию в кэше.

Еще одна причина, по которой следует избегать дублирования функциональности TCP на прикладном уровне, – потеря ответа сервера. Поскольку клиент не получил ответа, у него срабатывает таймер, и он посылает запрос повторно. При этом сервер должен дважды обработать один и тот же запрос, что может быть нежелательно. Представьте клиент, который «просит» сервер перевести деньги с одного банковского счета на другой. При использовании TCP логика повторных попыток реализована вне приложения, так что сервер вообще не определит, повторный ли это запрос.

Примечание *Здесь не рассматривается возможность сетевого сбоя или отказа одного из хостов. Подробнее это рассматривается в совете 9.*

Транзакционные приложения и некоторые проблемы, связанные с применением в них протоколов TCP и UDP, обсуждаются в RFC 955 [Braden 1985]. В этой работе автор отстаивает необходимость промежуточного протокола между ненадежным, но не требующим соединений UDP, и надежным, но зависящим от соединений TCP. Соображения, изложенные в этом RFC, легли в основу предложенного Брейденом протокола TCP Extensions for Transactions (T/TCP), который рассмотрен ниже.

Один из способов обеспечить надежность TCP без установления соединения – воспользоваться протоколом T/TCP. Это расширение TCP, позволяющее достичь для транзакций производительности, сравнимой с UDP, за счет отказа (как правило) от процедуры трехстороннего квитирования в ходе установления обычного TCP-соединения и сокращения фазы TIME-WAIT (совет 22) при разрыве соединения.

Обоснование необходимости T/TCP и идеи, лежащие в основе его реализации, описаны в RFC 1379 [Braden 1992a]. RFC 1644 [Braden 1994] содержит функциональную

спецификацию T/TCP, а также обсуждение некоторых вопросов реализации В работе [Stevens 1996] рассматривается протокол T/TCP, приводятся сравнение его производительности с UDP, изменения в API сокетов, необходимые для поддержки нового протокола, и его реализация в системе 4.4BSD.

К сожалению, протокол T/TCP не так широко распространен, хотя и реализован в FreeBSD, и существуют дополнения к ядру Linux 2.0.32 и SunOS 4.1.3.

Ричард Стивенс ведет страницу, посвященную T/TCP, на которой есть ссылки на различные посвященные этому протоколу ресурсы. Адрес Web-страницы – http://www.kohala.com/start/ttcp.html.

Резюме

Здесь рассмотрены шаги, необходимые для построения надежного протокола поверх UDP. Хотя и существуют приложения, например, DNS, в которых это сделано, но для корректного решения такой задачи необходимо практически заново реализовать TCP. Поскольку маловероятно, что реализованный на базе UDP протокол будет так же эффективен, как TCP, смысла в этом, как правило, нет.

В этом разделе также кратко обсуждается протокол T/TCP – модификация TCP для оптимизации транзакционных приложений. Хотя T/TCP решает многие проблемы, возникающие при использовании TCP для реализации транзакций, он пока не получил широкого распространения.

Совет 9. При всей надежности у TCP есть и недостатки

Как уже неоднократно отмечалось, TCP – надежный протокол. Иногда эту мысль выражают так: «TCP гарантирует доставку отправленных данных». Хотя эта формулировка часто встречается, ее следует признать исключительно неудачной.

Предположим, что вы отсоединили хост от сети в середине передачи данных. В таком случае TCP не сможет доставить оставшиеся данные А на практике происходят сбои в сети, аварии серверов, выключение машины пользователями без разрыва TCP-соединения. Все это мешает TCP доставить по назначению данные, переданные приложением.

Но еще важнее психологическое воздействие фразы о «гарантируемой TCP доставке» на излишне доверчивых сетевых программистов. Разумеется, никто не считает, что TCP обладает магической способностью доставлять данные получателю, невзирая на все препятствия. Вера в гарантированную доставку проявляется в небрежном программировании и, в частности, в легкомысленном отношении к проверке ошибок.

Что такое надежность

Прежде чем приступать к рассмотрению ошибок, с которыми можно столкнуться при работе с TCP, обсудим, что понимается под надежностью TCP. Если TCP не гарантирует доставку всех данных, то *что* же он гарантирует? Первый вопрос: *кому* дается гарантия? На рис. 2.18 показан поток данных от приложения А вниз к стеку TCP/IP на хосте А, через несколько промежуточных маршрутизаторов,

вверх к стеку TCP/IP на хосте В и, наконец, к приложению В. Когда TCP-сегмент покидает уровень TCP на хосте А, он «обертывается» в IP-датаграмму для передачи хосту на другой стороне. По пути он может пройти через несколько маршрутизаторов, но, как видно из рис. 2.18, маршрутизаторы не имеют уровня TCP, они лишь переправляют IP-датаграммы.

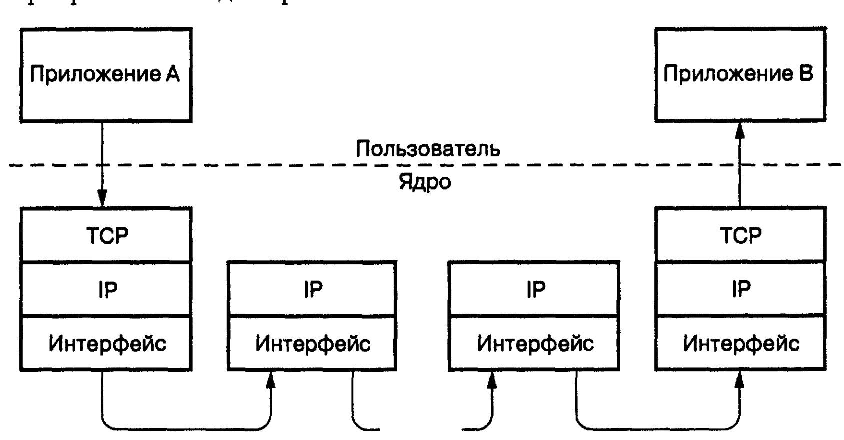


Рис. 2.18. Сеть с промежуточными маршрутизаторами

Примечание *Некоторые маршрутизаторы в действительности могут представлять собой компьютеры общего назначения, у которых есть полный стек TCP/IP, но и в этом случае при выполнении функций маршрутизации не задействуются ни уровень TCP, ни прикладной уровень.*

Поскольку известно, что протокол IP ненадежен, то первое место в тракте прохождения данных, в связи с которым имеет смысл говорить о гарантиях, – это уровень TCP хоста В. Когда сегмент оказывается на этом уровне, единственное, что можно сказать наверняка, – сегмент действительно прибыл. Он может быть запорчен, оказаться дубликатом, прийти не по порядку или оказаться неприемлемым еще по каким-то причинам. Обратите внимание, что отправляющий TCP не может дать никаких гарантий по поводу сегментов, доставленных принимающему TCP.

Однако принимающий TCP уже готов кое-что гарантировать отправляющему TCP, а именно – любые данные, которые он подтвердил с помощью сегмента ACK, а также все предшествующие данные, корректно дошли до уровня TCP. Поэтому отправляющий TCP может отбросить их копии, которые у него хранятся. Это *не означает*, что информация уже доставлена приложению или будет доставлена в будущем. Например, принимающий хост может аварийно остановиться сразу после посылки ACK, еще до того, как данные прочитаны приложением. Это стоит подчеркнуть особо: единственное подтверждение приема данных, которое находится в ведении TCP, – это вышеупомянутый сегмент ACK. Отправляющее приложение не может, полагаясь только на TCP, утверждать, что данные были благополучно прочитаны получателем. Как будет сказано далее, это одна из возможных ошибок при работе с TCP, о которых разработчик должен знать.

Второе место, в связи с которым имеет смысл говорить о гарантиях, – это само приложение В. Вы поняли, нет гарантий, что все данные, отправленные приложением А, дойдут до приложения В. Единственное, что TCP гарантирует приложению В, –доставленные данные пришли в правильном порядке и не испорчены.

Примечание *Неискаженность данных гарантируется лишь тем, что ошибку можно обнаружить с помощью контрольной суммы. Поскольку эта сумма представляет собой 16-разрядное дополнение до единицы суммы двойных байтов, то она способна обнаружить пакет ошибок в 15 бит или менее [Plummer 1978]. Предполагая равномерное распределение данных, вероятность принятия TCP ошибочного сегмента за правильный составляет не более $1 / (2^{16} - 1)$. Однако в работе [Stone et al. 1998] показано, что в реальных данных, встречающихся в сегментах TCP, частота ошибок, не обнаруживаемых с помощью контрольной суммы, при некоторых обстоятельствах может быть намного выше.*

Потенциальные ошибки

Вы уже видели одну из потенциальных ошибок при работе с TCP: не исключено, что подтвержденные TCP данные не дошли до приложения-получателя. Как и большинство других таких ошибок, это довольно редкая ситуация, и, даже если она встречается, последствия могут быть не очень печальными. Важно, чтобы программист знал об этой неприятности и предусматривал защиту при возможном нежелательном результате. Не думайте, что TCP обо всем позаботится сам, следует подумать об устойчивости приложения.

Защита от упомянутой ошибки очевидна. Если приложению-отправителю важно иметь информацию, что сообщение дошло до приложения-получателя, то получатель должен сам подтвердить факт приема. Часто такое подтверждение присутствует неявно. Например, если клиент запрашивает у сервера некоторые данные и сервер отвечает, то сам ответ – это подтверждение получения запроса. Один из возможных способов организации явных подтверждений обсуждается в совете 19.

Более сложный для клиента вопрос – что делать, если сервер не подтверждает приема? Это в основном зависит от конкретного приложения, поэтому готового решения не существует. Однако стоит отметить, что повторная посылка запроса не всегда годится; как говорилось в совете 8, вряд ли будет правильно дважды переводить одну сумму со счета на счет. В системах управления базами данных для решения такого рода проблем применяется протокол трехфазной фиксации. Подобный подход приемлем и для других приложений, гарантирующих, что операция выполняется «не более одного раза». Один из примеров – службы параллельности, фиксации и восстановления (concurrency, commitment, recovery – CCR) – это элемент прикладного сервиса в протоколах OSI. Протокол CCR обсуждается в работе [Jain and Agrawala 1993].

TCP – протокол сквозной передачи (end-to-end protocol), то есть он стремится обеспечить надежный транспортный механизм между двумя хостами одного ранга. Важно, однако, понимать, что конечные точки – это уровни TCP на обоих

хостах, а не приложения. Программы, которым нужны подтверждения на прикладном уровне, должны самостоятельно это определить.

Рассмотрим некоторые типичные ошибки. Пока между двумя хостами существует связь, TCP гарантирует доставку данных по порядку и без искажений. Ошибка может произойти только при разрыве связи. Из-за чего же связь может разорваться? Есть три причины:

- постоянный или временный сбой в сети;
- отказ принимающего приложения;
- аварийный сбой самого хоста на принимающем конце.

Каждое из этих событий по-разному отражается на приложении-отправителе.

Сбой в сети

Сбои в сети происходят по разным причинам: от потери связи с маршрутизатором или отрезком опорной сети до выдергивания из разъема кабеля локальной Ethernet-сети. Сбои, происходящие вне оконечных точек, обычно временные, поскольку протоколы маршрутизации спроектированы так, чтобы обнаруживать поврежденные участки и обходить их.

Примечание *Под оконечной точкой понимается локальная сеть или хост, на котором работает приложение.*

Если же ошибка возникает в оконечной точке, то неисправность будет существовать, пока ее не устранят.

Если промежуточный маршрутизатор не посылает ICMP-сообщение о том, что хост или сеть назначения недоступны, то ни само приложение, ни стек TCP/IP на том же хосте не смогут немедленно узнать о сбое в сети (совет 10). В этом случае у отправителя через некоторое время возникнет тайм-аут, и он повторно отправит неподтвержденные сегменты. Это будет продолжаться, пока отправляющий TCP не признает доставку невозможной, после чего он обрывает соединение и сообщает об ошибке. В системе BSD это произойдет после 12 безуспешных попыток (примерно 9 мин). При наличии у TCP ожидающего запроса на чтение операция возвращает ошибку, и переменная `errno` устанавливается в `ETIMEDOUT`. Если ожидающего запроса на чтение нет, то следующая операция записи завершится ошибкой. При этом либо будет послан сигнал `SIGPIPE`, либо (если этот сигнал перехвачен или игнорируется) в переменную `errno` записано значение `EPIPE`.

Если промежуточный маршрутизатор не может переправить далее IP-датаграмму, содержащую некоторый сегмент, то он посылает хосту-отправителю ICMP-сообщение о том, что сеть или хост назначения недоступны. В этом случае некоторые реализации возвращают в качестве кода ошибки значение ENETUNREACH или EHOSTUNREACH.

Отказ приложения

А теперь разберемся, что происходит, когда аварийно или как-либо иначе завершается приложение на другом конце соединения. Прежде всего следует понимать, что с точки зрения вашего приложения аварийное завершение другого конца не отличается от ситуации, когда приложение на том конце вызывает функцию

close (или closesocket, если речь идет о Windows), а затем exit. В обоих случаях TCP на другом конце посылает вашему TCP сегмент FIN. FIN выступает в роли признака конца файла и означает, что у отправившего его приложения нет больше данных для вас. Это не значит, что приложение на другом конце завершилось или не хочет *принимать* данные. Подробнее это рассмотрено в совете 16. Как приложение уведомляется о приходе FIN (и уведомляется ли вообще), зависит от его действий в этот момент. Для проработки возможных ситуаций напишем небольшую клиентскую программу, которая читает строку из стандартного входа, посылает ее серверу, читает ответ сервера и записывает его на стандартный выход. Исходный текст клиента приведен в листинге 2.21.

Листинг 2.21. TCP-клиент, который читает и выводит строки

tcprw.c

```
 1 #include "etcp.h"
 2 int main( int argc, char **argv )
 3 {
 4    SOCKET s;
 5    int rc;
 6    int len;
 7    char buf[ 120 ];
 8    INIT();
 9    s = tcp_client( argv[ 1 ], argv[ 2 ] );
10    while ( fgets( buf, sizeof( buf ), stdin ) != NULL )
11    {
12       len = strlen( buf );
13       rc = send( s, buf, len, 0 );
14       if ( rc < 0 )
15          error( 1, errno, "ошибка вызова send" );
16       rc = readline( s, buf, sizeof( buf ) );
17       if ( rc < 0 )
18          error( 1, errno, "ошибка вызова readline" );
19       else if ( rc == 0 )
20          error( 1, 0, "сервер завершил работу\n" );
21       else
22          fputs( buf, stdout );
23    }
24    EXIT( 0 );
25 }
```

tcprw.c

8-9 Инициализируем приложение как TCP-клиент и соединяемся с указанными в командной строке сервером и портом.

10-15 Читаем строки из стандартного входа и посылаем их серверу, пока не встретится конец файла.

16-20 После отправки данных серверу читается строка ответа. Функция readline получает строку, считывая данные из сокета до символа новой строки. Текст этой функции приведен в листинге 2.32 в совете 11. Если

readline обнаруживает ошибку или возвращает признак конца файла (совет 16), то печатаем диагностическое сообщение и завершаем работу.

22 В противном случае выводим строку на stdout.

Для тестирования клиента напишем сервер, который читает в цикле строки, поступающие от клиента, и возвращает сообщения о количестве полученных строк. Для имитации задержки между приемом сообщения и отправкой ответа сервер пять секунд «спит». Код сервера приведен в листинге 2.22.

Листинг 2.22. Сервер, подсчитывающий сообщения

count.c

```
#include "etcp.h"
int main( int argc, char **argv )
   {
   SOCKET s;
   SOCKET s1;
   int rc;
   int len;
   int counter = 1;
   char buf[ 120 ];

   INIT();
   s = tcp_server( NULL, argv[ 1 ] );
   s1 = accept( s, NULL, NULL );
   if ( !isvalidsock( s1 ) )
      error( 1, errno, "ошибка вызова accept" );
   while ( ( rc = readline( s1, buf, sizeof( buf ) ) ) > 0 )
   {
      sleep( 5 );
      len = sprintf( buf, "получено сообщение %d\n", counter++ );
      rc = send( s1, buf, len, 0 );
      if ( rc < 0 )
         error( 1, errno, "ошибка вызова send" );
   }
   EXIT( 0 );
}
```

count.c

Чтобы увидеть, что происходит при крахе сервера, сначала запустим сервер и клиент в различных окнах на машине bsd.

```
bsd: $ tcprw localhost 9000
hello
получено сообщение 1  Это печатается после пятисекундной задержки.
                      Здесь сервер был остановлен.
hello again
tcprw: ошибка вызова readline: Connection reset by peer (54)
bsd: $
```

Серверу посылается одно сообщение, и через 5 с приходит ожидаемый ответ. Останавливаете серверный процесс, моделируя аварийный отказ. На стороне

клиента ничего не происходит. Клиент блокирован в вызове fgets, а протокол TCP не может передать клиенту информацию о том, что от другого конца получен конец файла (FIN). Если ничего не делать, то клиент так и останется блокированным в ожидании ввода и не узнает о завершении сеанса сервера.

Затем вводите новую строку. Клиент немедленно завершает работу с сообщением о том, что хост сервера сбросил соединение. Вот что произошло: функция fgets вернула управление клиенту, которому все еще неизвестно о приходе признака конца файла от сервера. Поскольку ничто не мешает приложению посылать данные после прихода FIN, TCP клиента попытался послать серверу вторую строку. Когда TCP сервера получил эту строку, он послал в ответ сегмент RST (сброс), поскольку соединения уже не существует, – сервер завершил сеанс. Когда клиент вызывает readline, ядро возвращает ему код ошибки ECONNRESET, сообщая тем самым о получении извещения о сбросе. На рис. 2.19 показана хронологическая последовательность этих событий.

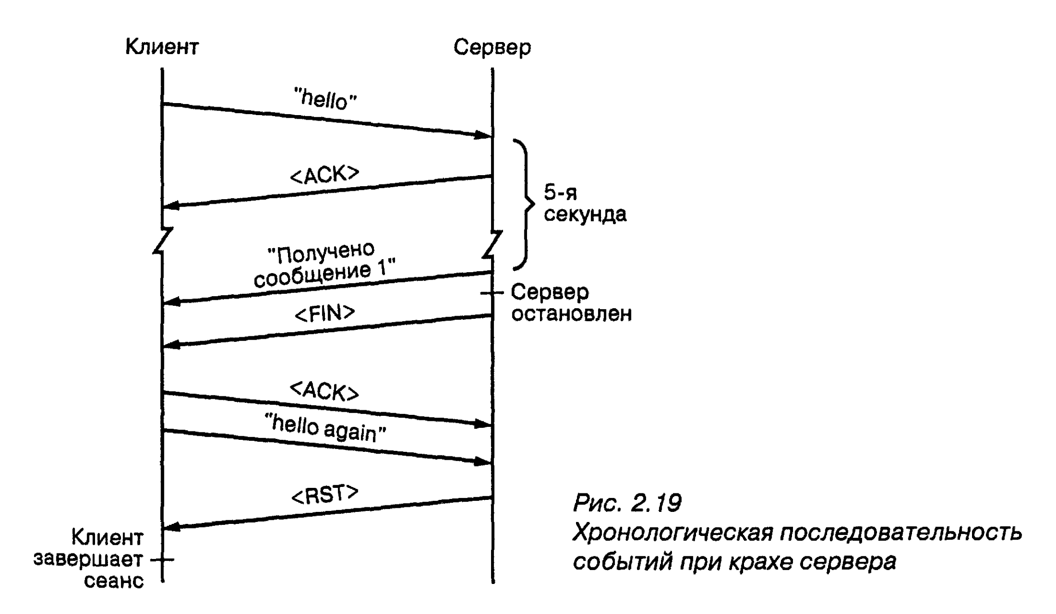


Рис. 2.19 Хронологическая последовательность событий при крахе сервера

А теперь рассмотрим ситуацию, когда сервер «падает», не успев закончить обработку запроса и ответить. Снова запустите сервер и клиент в разных окнах на машине bsd.

```
bsd: $ tcprw localhost 9000
hello
                                  Здесь сервер был остановлен.
tcprw: сервер завершил работу
bsd: $
```

Посылаете строку серверу, а затем прерываете его работу до завершения вызова sleep. Тем самым имитируется крах сервера до завершения обработки запроса. На

этот раз клиент немедленно получает сообщение об ошибке, говорящее о завершении сервера. В этом примере клиент в момент прихода FIN блокирован в вызове `readline` и TCP может уведомить `readline` сразу, как только будет получен конец файла. Хронологическая последовательность этих событий изображена на рис. 2.20.

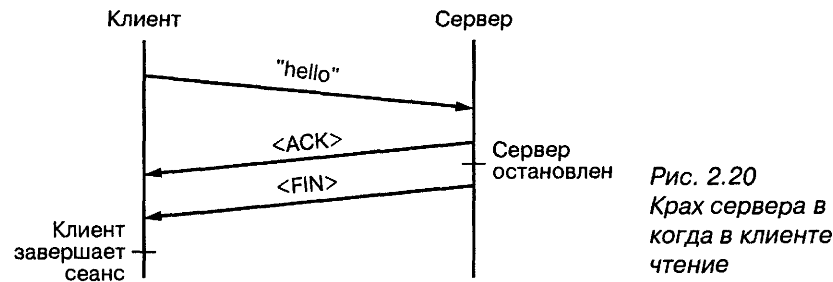


Рис. 2.20 Крах сервера в момент, когда в клиенте происходит чтение

Ошибка также может произойти, если игнорировать извещение о сбросе соединения и продолжать посылать данные. Чтобы промоделировать эту ситуацию, следует изменить обращение к функции `error` после `readline` – вывести диагностическое сообщение, но не завершаться. Для этого достаточно вместо строки 17 в листинге 2.21 написать

```
error( 0, errno, "ошибка при вызове readline" );
```

Теперь еще раз надо прогнать тест:

```
bsd: $ tcprw localhost 9000
hello
получено сообщение 1
                                        Здесь сервер был остановлен.
hello again
tcprw: ошибка вызова readline: Connection reset by peer (54)
                                        Клиент игнорирует ошибку, но
                                        TCP уже разорвал соединение.
hello for the last time
Broken pipe                             Клиент получает сигнал SIGPIPE
                                        и завершает работу.
bsd: $
```

Когда вводится вторая строка, клиент, как и раньше, немедленно извещает об ошибке (соединение сброшено сервером), но не завершает сеанс. Он еще раз обращается к `fgets`, чтобы получить очередную строку для отправки серверу. Но стоит внести эту строку, как клиент тут же прекращает работу, и командный интерпретатор сообщает, что выполнение было прервано сигналом `SIGPIPE`. В этом случае при втором обращении к `send`, как и прежде, TCP послал RST, но вы не обратили на него внимания. Однако после получения RST клиентский TCP разорвал соединение, поэтому при попытке отправить третью строку он немедленно завершает клиента, посылая ему сигнал `SIGPIPE`. Хронология такая же, как на рис. 2.19. Разница лишь в том, что клиент «падает» при попытке записи, а не чтения.

Правильно спроектированное приложение, конечно, не игнорирует ошибки, но такая ситуация может иметь место и в корректно написанных программах. Предположим, что приложение выполняет подряд несколько операций записи без промежуточного чтения. Типичный пример – FTP. Если приложение на другом конце «падает», то TCP посылает сегмент FIN. Поскольку данная программа только пишет, но не читает, в ней не содержится информация о получении этого FIN. При отправке следующего сегмента TCP на другом конце вернет RST. А в программе опять не будет никаких сведений об этом, так как ожидающей операции чтения нет. При второй попытке записи после краха отвечающего конца программа получит сигнал `SIGPIPE`, если этот сигнал перехвачен или игнорируется – код ошибки `EPIPE`.

Такое поведение вполне типично для приложений, выполняющих многократную запись без чтения, поэтому надо отчетливо представлять себе последствия. Приложение уведомляется только после *второй* операции отправки данных завершившемуся партнеру. Но, так как предшествующая операция привела к сбросу соединения, посланные ей данные были потеряны.

Поведение зависит от соотношения времен. Например, если снова прогнать первый тест, запустив сервер на машине `sparc`, а клиента – на машине `bsd`, то получается следующее:

```
bsd: $ tcprw localhost 9000
hello
получено сообщение 1          Это печатается после пятисекундной
                              задержки.
                              Здесь сервер был остановлен.
hello again
tcprw: сервер завершил работу
bsd: $
```

На этот раз клиент обнаружил конец файла, посланный в результате остановки сервера. RST по-прежнему генерируется при отправке второй строки, но из-за задержек в сети клиент успевает вызвать `readline` и обнаружить конец файла еще до того, как хост `bsd` получит RST. Если вставить между строками 14 и 15 в листинге 2.21 строчку

```
sleep( 1 );
```

с целью имитировать обработку на клиенте или загруженность системы, то получится тот же результат, что и при запуске клиента и сервера на одной машине.

Крах хоста на другом конце соединения

И последняя ошибка, которую следует рассмотреть, – это аварийный останов хоста на другом конце. Ситуация отличается от краха хоста, поскольку TCP на другом конце не может с помощью сегмента FIN проинформировать программу о том, что ее партнер уже не работает.

Пока хост на другом конце не перезагрузят, ситуация будет выглядеть как сбой в сети – TCP удаленного хоста не отвечает. Как и при сбое в сети, TCP продолжает повторно передавать неподтвержденные сегменты. Но в конце концов,

если удаленный хост так и не перезагрузится, то TCP вернет приложению код ошибки ETIMEDOUT.

А что произойдет, если удаленный хост перезагрузится до того, как TCP прекратит попытки и разорвет соединение? Тогда повторно передаваемые вами сегменты дойдут до перезагрузившегося хоста, в котором нет никакой информации о старых соединениях. В таком случае спецификация TCP [Postel 1981b] требует, чтобы принимающий хост послал отправителю RST. В результате отправитель оборвет соединение, и приложение либо получит код ошибки ECONNRESET (если есть ожидающее чтение), либо следующая операция записи закончится сигналом SIGPIPE или ошибкой EPIPE.

Резюме

В этом разделе дано объяснение понятию «надежность TCP». Вы узнали, что не существует гарантированной доставки, и при работе с TCP могут встретиться разнообразные ошибки. Ни одна из этих ошибок не фатальна, но вы должны быть готовы к их обработке.

Совет 10. Помните, что TCP не выполняет опрос соединения

Программисты, приступающие к изучению семейства протоколов TCP/IP, но имеющие опыт работы с другими сетевыми технологиями, часто удивляются, что TCP не посылает приложению немедленного уведомления о потере связи. Поэтому некоторые даже считают, что TCP не пригоден в качестве универсальной технологии обмена данными между приложениями. В этом разделе разъясняются причины отсутствия у TCP средств для уведомления, достоинства и недостатки такого подхода и способы обнаружения потери связи прикладным программистом.

Как вы узнали в совете 9, сетевой сбой или крах системы могут прервать сообщение между хостами, но приложения на обоих концах соединения «узнают» об этом не сразу. Приложение-отправитель остается в неведении до тех пор, пока TCP не исчерпает все попытки. Это продолжается довольно долго, в системах на базе BSD – примерно 9 мин. Если приложение не посылает данные, то оно может вообще не получить информации о потере связи. Например, приложение-сервер ожидает, пока клиент не обратится со следующим запросом. Но, поскольку у клиента нет связи с сервером, следующий запрос не придет. Даже когда TCP на стороне клиента прекратит свои попытки и оборвет соединение, серверу об этом будет ничего не известно.

Другие коммуникационные протоколы, например SNA или X.25, извещают приложение о потере связи. Если имеется нечто более сложное, чем простая двухточечная выделенная линия, то необходим протокол опроса, который постоянно проверяет наличие абонента на другом конце соединения. Это может быть сообщение типа «есть что-нибудь для отправки?» или скрыты е фреймы, посылаемые в фоновом режиме для непрерывного наблюдения за состоянием виртуального канала. В любом случае, за эту возможность приходится расплачиваться пропускной способностью сети. Каждое такое опрашивающее сообщение потребляет сетевые ресурсы, которые могли бы использоваться для увеличения полезной нагрузки.

Очевидно, одна из причин, по которым TCP не уведомляет о потере связи немедленно, – это нежелание жертвовать полосой пропускания. Большинству приложений немедленное уведомление и не нужно. Приложение, которому действительно необходимо срочно узнавать о недоступности другого конца, может реализовать для этой цели собственный механизм. Далее будет показано, как это сделать.

Есть и философское возражение против встраивания в TCP/IP механизма немедленного уведомления. Один из фундаментальных принципов, заложенных при проектировании TCP/IP, – это принцип «оконечного разума» [Saltzer et al. 1984]. В применении к сетям упрощенно подразумевается следующее. «Интеллекту» нужно находиться как можно ближе к оконечным точкам соединения, а сама сеть должна быть относительно «неинтеллектуальной». Именно поэтому TCP обрабатывает ошибки самостоятельно, не полагаясь на сеть. Как сказано в совете 1, протокол IP (значит, и построенный на его основе TCP) делает очень мало предположений о свойствах физической сети. Относительно мониторинга наличия связи между приложениями этот принцип означает, что такой механизм должен реализовываться теми приложениями, которым это необходимо, а не предоставляться всем приложениям без разбора. В работе [Huitema 1995] принцип «оконечного разума» интересно обсуждается в применении к Internet.

Однако веская причина отсутствия у TCP средств для немедленного уведомления о потере соединения связана с одной из главных целей его проектирования: способностью поддерживать связь при наличии сбоев в сети. Протокол TCP – это результат исследований, проведенных при финансовой поддержке Министерства обороны США, с целью создания надежной технологии связи между компьютерами. Такая технология могла бы функционировать даже в условиях обрывов сетей из-за военных действий или природных катастроф. Часто сетевые сбои быстро устраняются или маршрутизаторы находят другой маршрут для соединения. Допуская временную потерю связи, TCP часто может справиться со сбоями, не ставя об этом в известность приложения.

Недостаток такого подхода в том, что код, отслеживающий наличие связи, необходимо встраивать в каждое приложение (которому это нужно), а непродуманная реализация может привести к ненужному расходу ресурсов или как-то иначе повредить пользователям. Но и в этом случае при встраивании мониторинга в приложение можно осуществить тонкую настройку алгоритма, чтобы он удовлетворял нуждам приложения и по возможности естественно интегрировался с прикладным протоколом.

Механизм контролеров

В действительности протокол TCP обладает механизмом обнаружения мертвых соединений – так называемыми контролерами (keep-alive). Но, как вы вскоре увидите, для приложений подобный механизм часто бесполезен. Если приложение его активирует, то TCP посылает на другой конец специальный сегмент, когда по соединению в течение некоторого времени не передавались данные. Если хост на другом конце доступен и приложение там все еще работает, то TCP отвечает сегментом ACK. В этом случае TCP, пославший контролера, сбрасывает время простоя в нуль; приложение не получает извещения о том, что имел место обмен информацией.

Если хост на другом конце работает, а приложение – нет, то TCP посылает в ответ сегмент RST. А TCP, отправивший контролер, разрывает соединение и возвращает приложению код ECONNRESET. Обычно так бывает после перезагрузки удаленного хоста, поскольку, как говорилось в совете 9, если бы завершилось всего лишь приложение на другом конце, то TCP послал сегмент FIN.

Если удаленный хост не посылает в ответ ни ACK, ни RST, то TCP продолжает посылать контролеров, пока не получит сведений, что хост недоступен. В этот момент он разрывает соединение и возвращает приложению код ETIMEDOUT либо, если маршрутизатор прислал ICMP-сообщение о недоступности хоста или сети, соответственно код EHOSTUNREACH или ENETUNREACH.

Первая проблема, с которой сталкиваются приложения, нуждающиеся в немедленном уведомлении, при попытке воспользоваться механизмом контролеров, – это длительность временных интервалов. В соответствии с RFC 1122 [Braden 1989], если TCP реализует механизм контролеров, то по умолчанию время простоя должно быть не менее двух часов. И только после этого можно посылать контролеров. Затем, поскольку ACK, посланный удаленным хостом, доставляется ненадежно, процесс отправки контролеров необходимо несколько раз повторить; и лишь тогда можно разрывать соединение. В системе 4.4BSD отправляется девять контролеров с интервалом 75 с.

Примечание *Точные величины – деталь реализации. В RFC 1122 не говорится о том, сколько и с каким интервалом нужно посылать контролеры, прежде чем разорвать соединение. Утверждается лишь, что реализация не должна интерпретировать отсутствие ответа на посылку одного контролера как индикатор прекращения соединения.*

Таким образом, в реализациях на основе BSD для обнаружения потери связи потребуется 2 ч 11 мин 15 с. Этот срок приобретает смысл, если вы понимаете, что назначение контролеров – освободить ресурсы, занятые уже несуществующими соединениями. Такое возможно, например, если клиент соединяется с сервером, а затем хост клиента неожиданно отключается. Без механизма дежурных серверу пришлось бы ждать следующего запроса от клиента вечно, поскольку он не получит FIN.

Примечание *Эта ситуация очень распространена из-за ошибок пользователей персональных компьютеров, которые просто выключают компьютер или модем, не завершив корректно работающие приложения.*

В некоторых реализациях разрешено изменять один или оба временных интервала, но это всегда распространяется на систему в целом. Иными словами, изменение затрагивает *все* TCP-соединения, установленные данной системой. Это и есть основная причина, по которой механизм контролеров почти бесполезен в качестве средства мониторинга связи. Период, выбранный по умолчанию, слишком велик, а если его сократить, то контролеры перестанут выполнять свою исходную задачу – обнаруживать давно «зависшие» соединения.

В последней версии стандарта POSIX появилась новейшая опция сокета TCP_KEEPALIVE, которая позволяет устанавливать временной интервал для отдельного соединения, но пока она не получила широкого распространения.

Еще одна проблема, связанная с механизмом контролеров, состоит в том, что он не просто обнаруживает «мертвые» соединения, а еще и разрывает их независимо от того, допускает ли это приложение.

Пульсация

Задача проверки наличия соединения, неразрешимая с помощью механизма контролеров, легко решается путем реализации аналогичного механизма в самом приложении. Оптимальный метод зависит от приложения. Здесь вы можете полнее оценить гибкость, которая может быть достигнута при реализации на прикладном уровне. В качестве примеров рассмотрим два крайних случая:

- клиент и сервер обмениваются сообщениями разных типов, каждое из которых имеет заголовок, идентифицирующий тип сообщения;
- приложение передает данные в виде потока байтов без разбиения на записи.

Первый случай сравнительно несложен. Вводится новый тип сообщения MSG_HEARTBEAT. Получив такое сообщение, приложение возвращает его отправителю. Такой способ предоставляет большую свободу. Проверять наличие связи могут одна или обе стороны, причем только одна действительно посылает контрольное сообщение-пульс.

Сначала рассмотрим заголовочный файл (листинг 2.23), который используют как клиент, так и сервер.

Листинг 2.23. Заголовочный файл для реализации механизма пульсации

heartbeat.h

```
 1 #ifndef __HEARTBEAT_H__
 2 #define __HEARTBEAT_H__
 3 #define MSG_TYPE1      1   /* Сообщение прикладного уровня. */
 4 #define MSG_TYPE2      2   /* Еще одно. */
 5 #define MSG_HEARTBEAT 3    /* Сообщение-пульс. */
 6 typedef struct             /* Структура сообщения. */
 7 {
 8 u_int32_t type;            /* MSG_TYPE1, ... */
 9 char data[ 2000 ];
10 } msg_t;
11 #define T1          60 /* Время простоя перед отправкой пульса. */
12 #define T2          10 /* Время ожидания ответа. */
13 #endif  /* __HEARTBEAT_H__ */
```

heartbeat.h

3-5 С помощью этих констант определяются различные типы сообщений, которыми обмениваются клиент и сервер. Для данного примера нужно только сообщение MSG_HEARTBEAT.

6-10 Здесь определяется структура сообщений, которыми обмениваются клиент и сервер. Здесь представляет интерес только поле `type`. Реальное приложение могло бы подстроить эту структуру под свои возможности. Подробнее это рассматривается в замечаниях к листингу 2.15 о смысле типа `u_int32_t` и об опасности предположений о способе упаковки структур.

11 Данная константа определяет, сколько времени может простаивать соединение, прежде чем приложение начнет посылать контрольные сообщения-пульсы. Здесь произвольно выбрано 60 с, реальное же приложение должно подобрать значение, наиболее соответствующее его потребностям и виду сети.

12 Эта константа определяет, сколько времени клиент будет ждать ответа на контрольное сообщение.

В листинге 2.24 приведен текст клиента, который инициирует посылку контрольных сообщений. Такой выбор абсолютно произволен, в качестве инициатора можно было выбрать сервер.

Листинг 2.24. Клиент, посылающий контрольные сообщения-пульсы

hb_client.c

```
#include "etcp.h"
#include "heartbeat.h"
int main( int argc, char **argv )
{
   fd_set allfd;
   fd_set readfd;
   msg_t msg;
   struct timeval tv;
   SOCKET s;
   int rc;
   int heartbeats = 0;
   int cnt = sizeof( msg );
   INIT();
   s = tcp_client( argv[ 1 ], argv[ 2 ] );
   FD_ZERO( &allfd );
   FD_SET( s, &allfd );
   tv.tv_sec = T1;
   tv.tv_usec = 0;
   for ( ;; )
   {
      readfd = allfd;
      rc = select( s + 1, &readfd, NULL, NULL, &tv );
      if ( rc < 0 )
         error( 1, errno, "ошибка вызова select" );
      if ( rc == 0 )  /* Произошел тайм-аут. */
      {
         if ( ++heartbeats > 3 )
         error( 1, 0, "соединения нет\n" );
```

```
29        error( 0, 0, "посылаю пульс #%d\n", heartbeats );
30        msg.type = htonl( MSG_HEARTBEAT );
31        rc = send( s, ( char * )&msg, sizeof( msg ), 0 );
32        if ( rc < 0 )
33           error( 1, errno, "ошибка вызова send" );
34        tv.tv_sec = T2;
35        continue;
36     }
37     if ( !FD_ISSET( s, &readfd ) )
38        error( 1, 0, "select вернул некорректный сокет\n" );
39     rc = recv( s, ( char * )&msg + sizeof( msg ) - cnt,
40        cnt, 0 );
41     if ( rc == 0 )
42        error( 1, 0, "сервер закончил работу\n" );
43     if ( rc < 0 )
44        error( 1, errno, "ошибка вызова recv" );
45     heartbeats = 0;
46     tv.tv_sec = T1;
47     cnt -= rc;              /* Встроенный readn. */
48     if ( cnt > 0 )
49        continue;
50     cnt = sizeof( msg );
51     /* Обработка сообщения. */
52     }
53  }
```

hb_client.c

Инициализация

13-14 Выполняем стандартную инициализацию и соединяемся с сервером, адрес и номер порта которого заданы в командной строке.

15-16 Задаем маску для системного вызова `select`, в которой выбран ваш сокет.

17-18 Взводим таймер на T1 секунд. Если за это время не было получено никакого сообщения, то `select` вернет управление с индикацией срабатывания таймера.

21-22 Устанавливаем маску, выбирающую сокет, из которого читаем, после чего система блокирует программу в вызове `select`, пока не поступят данные либо не сработает таймер.

Обработка тайм-аута

27-28 Если послано подряд более трех контрольных пульсов и не получено ответа, то считается, что соединение «мертво». В этом примере просто завершаем работу, но реальное приложение могло бы предпринять более осмысленные действия.

29-33 Если максимальное число последовательных контрольных пульсов не достигнуто, посылается новый пульс.

34-35 Устанавливаем таймер на T2 секунд. Если за это время не получен ответ, то либо отправляется новый пульс, либо соединение признается «мертвым» в зависимости от значения переменной `heartbeats`.

Обработка сообщения

37-38 Если `select` вернул сокет, отличный от соединенного с сервером, то завершаемся с сообщением о фатальной ошибке.

39-40 Вызываем `recv` для чтения одного сообщения. Эти строки, а также следующий за ними код, изменяющий значение переменной `cnt`, – не что иное, как встроенная версия функции `readn`. Она не может быть вызвана напрямую, поскольку заблокировала бы весь процесс на неопределенное время, нарушив тем самым работу механизма пульсации.

41-44 Если получаем признак конца файла или ошибку чтения, выводим диагностическое сообщение и завершаем сеанс.

45-46 Поскольку только что получен ответ от сервера, сбрасывается счетчик пульсов в 0 и переустанавливается таймер на T1 секунд.

47-50 Эти строки завершают встроенный вариант `readn`. Уменьшаем переменную `cnt` на число, равное количеству только что прочитанных байт. Если прочитано не все, то следует повторить цикл с вызова `select`. В противном случае заносится в `cnt` полная длина сообщения и завершается обработка только что принятого сообщения.

Листинг 2.25 содержит текст сервера для этого примера. Здесь предполагается что сервер также будет следить за состоянием соединения, но это не обязательно.

Листинг 2.25. Сервер, отвечающий на контрольные сообщения-пульсы

hb_server.c

```
#include "etcp.h"
#include "heartbeat.h"

int main( int argc, char **argv )
{
    fd_set allfd;
    fd_set readfd;
    msg_t msg;
    struct timeval tv;
    SOCKET s;
    SOCKET s1;
    int rc;
    int missed_heartbeats = 0;
    int cnt = sizeof( msg );

    INIT();
    s = tcp_server( NULL, argv[ 1 ] );
    s1 = accept( s, NULL, NULL );
    if ( !isvalidsock( s1 ) )
      error( 1, errno, "ошибка вызова accept" );
    tv.tv_sec = T1 + T2;
    tv.tv_usec = 0;
    FD_ZERO( &allfd );
    FD_SET( s1, &allfd );
    for ( ;; )
    {
```

```
        readfd = allfd;
        rc = select( s1 + 1, &readfd, NULL, NULL, &tv );
        if ( rc < 0 )
            error( 1, errno, "ошибка вызова select" );
        if ( rc == 0 )  /* Произошел тайм-аут. */
        {
            if ( ++missed_heartbeats > 3 )
                error( 1, 0, "соединение умерло\n" );
            error( 0, 0, "пропущен пульс #%d\n",
                missed_heartbeats );
            tv.tv_sec = T2;
            continue;
        }
        if ( !FD_ISSET( s1, &readfd ) )
            error( 1, 0, "select вернул некорректный сокет\n" );
        rc = recv( s1, ( char * )&msg + sizeof( msg ) - cnt,
            cnt, 0 );
        if ( rc == 0 )
            error( 1, 0, "клиент завершил работу\n" );
        if ( rc < 0 )
            error( 1, errno, "ошибка вызова recv" );
        missed_heartbeats = 0;
        tv.tv_sec = T1 + T2;
        cnt -= rc;  /* Встроенный readn. */
        if ( cnt > 0 )
            continue;
        cnt = sizeof( msg );
        switch ( ntohl( msg.type ) )
        {
            case MSG_TYPE1 :
                /* Обработать сообщение типа TYPE1. */
                break;

            case MSG_TYPE2 :
                /* Обработать сообщение типа TYPE2. */
                break;

            case MSG_HEARTBEAT :
                rc = send( s1, ( char * )&msg, sizeof( msg ), 0 );
                if ( rc < 0 )
                    error( 1, errno, "ошибка вызова send" );
                break;

            default :
                error( 1, 0, "неизвестный тип сообщения (%d)\n",
                    ntohl( msg.type ) );
        }
    }
    EXIT( 0 );
    }
```

hb_server.c

Инициализация

14-18 Выполняем стандартную инициализацию и принимаем соединение от клиента.

19-20 Взводим таймер на T1 + T2 секунд. Поскольку клиент посылает пульс после T1 секунд неактивности, следует подождать немного больше – на T2 секунд.

21-22 Инициализируем маску для `select`, указывая в ней соединенный сокет, из которого происходит чтение.

25-28 Вызываем `select` и проверяем возвращенное значение.

Обработка тайм-аута

31-32 Если пропущено более трех пульсов подряд, то соединение считается «мертвым» – работа завершается. Как и клиент, реальный сервер мог бы предпринять в этом случае более осмысленные действия.

35 Взводим таймер на T2 секунд. К этому моменту клиент должен был бы посылать пульсы каждые T2 секунд, так что если за это время ничего не получено, то необходимо увеличить счетчик пропущенных пульсов.

Обработка сообщения

38-39 Производим ту же проверку корректности сокета, что и в клиенте.

40-41 Как и в клиенте, встраиваем код функции `readn`.

42-45 Если `recv` возвращает признак конца файла или код ошибки, то печатаем диагностическое сообщение и выходим.

46-47 Поскольку только что получено сообщение от клиента, соединение все еще живо, так что сбрасываем счетчик пропущенных пульсов в нуль и взводим таймер на T1 + T2 секунд.

48-51 Этот код, такой же, как в клиенте, завершает встроенную версию `readn`.

60-64 Если это сообщение-пульс, то возвращаем его клиенту. Когда клиент получит сообщение, обе стороны будут знать, что соединение еще есть.

Для тестирования этих программ запустим программу `hb_server` на машине `sparc`, а программу `hb_client` – на машине `bsd`. После того как клиент соединится с сервером, отключим `sparc` от сети. Вот что при этом будет напечатано:

```
sparc: $ hb_server 9000
hb_server: пропущен пульс #1
hb_server: пропущен пульс #2
hb_server: пропущен пульс #3
hb_server: соединения нет
sparc: $
```

```
bsd: $ hb_client sparc 9000
hb_client: посылаю пульс #1
hb_client: посылаю пульс #2
hb_client: посылаю пульс #3
hb_client: соединения нет
bsd: $
```

Еще один пример пульсации

Использованная в предыдущем примере модель не совсем пригодна в ситуации, когда одна сторона посылает другой поток данных, не разбитый на сообщения.

Проблема в том, что посланный пульс оказывается частью потока, поэтому его придется явно выискивать и, возможно, даже экранировать (совет 6). Чтобы избежать сложностей, следует воспользоваться другим подходом.

Идея в том, чтобы использовать для контрольных пульсов отдельное соединение. На первый взгляд, кажется странной возможность контролировать одно соединение с помощью другого. Но помните, что делается попытка обнаружить крах хоста на другом конце или разрыв в сети. Если это случится, то пострадают оба соединения. Задачу можно решить несколькими способами. Традиционный способ – создать отдельный поток выполнения (thread) для управления пульсацией. Можно также применить универсальный механизм отсчета времени, который разработан в совете 20. Однако, чтобы не вдаваться в различия между API потоков на платформе Win32 и библиотекой PThreads в UNIX, модифицируем написанный для предыдущего примера код с использованием системного вызова `select`.

Новые версии клиента и сервера очень похожи на исходные. Основное различие состоит в логике работы `select`, который теперь должен следить за двумя сокетами, а также в дополнительном коде для инициализации еще одного соединения. После соединения клиента с сервером, клиент посылает ему номер порта, по которому отслеживается пульсация сервера. Это напоминает то, что делает FTP-сервер, устанавливая соединение для обмена данными с клиентом.

Примечание *Может возникнуть проблема, если для преобразования частных сетевых адресов в открытые используется механизм NAT (совет 3). В отличие от ситуации с FTP программное обеспечение NAT не имеет информации, что нужно подменить указанный номер порта преобразованным. В таком случае самый простой путь – выделить приложению второй хорошо известный порт.*

Начнем с логики инициализации и установления соединения на стороне клиента (листинг 2.26).

Листинг 2.26. Код инициализации и установления соединения на стороне клиента

hb_client2.c

```
#include "etcp.h"
#include "heartbeat.h"
int main( int argc, char **argv )
{
  fd_set allfd;
  fd_set readfd;
  char msg[ 1024 ];
  struct timeval tv;
  struct sockaddr_in hblisten;
  SOCKET sdata;
  SOCKET shb;
  SOCKET slisten;
  int rc;
  int hblistenlen = sizeof( hblisten );
```

```
15 int heartbeats = 0;
16 int maxfd1;
17 char hbmsg[ 1 ];
18 INIT();
19 slisten = tcp_server( NULL, "0" );
20 rc = getsockname( slisten, ( struct sockaddr * )&hblisten,
21   &hblistenlen );
22 if ( rc )
23   error( 1, errno, "ошибка вызова getsockname" );
24 sdata = tcp_client( argv[ 1 ], argv[ 2 ] );
25 rc = send( sdata, ( char * )&hblisten.sin_port,
26   sizeof( hblisten.sin_port ), 0 );
27 if ( rc < 0 )
28   error( 1, errno, "ошибка при посылке номера порта" );
29 shb = accept( slisten, NULL, NULL );
30 if ( !isvalidsock( shb ) )
31   error( 1, errno, "ошибка вызова accept" );
32 FD_ZERO( &allfd );
33 FD_SET( sdata, &allfd );
34 FD_SET( shb, &allfd );
35 maxfd1 = ( sdata > shb ? sdata: shb ) + 1;
36 tv.tv_sec = T1;
37 tv.tv_usec = 0;
```

hb_client2.c

Инициализация и соединение

19-23 Вызываем функцию `tcp_server` с номером порта 0, таким образом заставляя ядро выделить эфемерный порт (совет 18). Затем вызываем `getsockname`, чтобы узнать номер этого порта. Это делается потому, что с данным сервером ассоциирован только один хорошо известный порт.

24-28 Соединяемся с сервером и посылаем ему номер порта, с которым он должен установить соединение для посылки сообщений-пульсов.

29-31 Вызов `accept` блокирует программу до тех пор, пока сервер не установит соединение для пульсации. В промышленной программе, наверное, стоило бы для этого вызова взвести таймер, чтобы программа не «зависла», если сервер не установит соединения. Можно также проверить, что соединение для пульсации определил именно тот сервер, который запрашивался в строке 24.

32-37 Инициализируем маски для `select` и взводим таймер.

Оставшийся код клиента показан в листинге 2.27. Здесь вы видите обработку содержательных сообщений и контрольных пульсов.

Листинг 2.27. Обработка сообщений клиентом

hb_client2.c

```
38 for ( ;; )
39 {
40   readfd = allfd;
```

```
41    rc = select( maxfd1, &readfd, NULL, NULL, &tv );
42    if ( rc < 0 )
43       error( 1, errno, "ошибка вызова select" );
44    if ( rc == 0 )  /* Произошел тайм-аут. */
45    {
46       if ( ++heartbeats > 3 )
47          error( 1, 0, "соединения нет\n" );
48       error( 0, 0, "посылаю пульс #%d\n", heartbeats );
49       rc = send( shb, "", 1, 0 );
50       if ( rc < 0 )
51          error( 1, errno, "ошибка вызова send" );
52       tv.tv_sec = T2;
53       continue;
54    }
55    if ( FD_ISSET( shb, &readfd ) )
56    {
57       rc = recv( shb, hbmsg, 1, 0 );
58       if ( rc == 0 )
59          error( 1, 0, "сервер закончил работу (shb)\n" );
60       if ( rc < 0 )
61          error( 1, errno, "ошибка вызова recv для сокета shb"
);
62    }
63    if ( FD_ISSET( sdata, &readfd ) )
64    {
65       rc = recv( sdata, msg, sizeof( msg ), 0 );
66       if ( rc == 0 )
67          error( 1, 0, "сервер закончил работу (sdata)\n" );
68       if ( rc < 0 )
69          error( 1, errno, "ошибка вызова recv" );
70       /* Обработка данных. */
71    }
72    heartbeats = 0;
73    tv.tv_sec = T1;
74    }
75 }
```

hb_client2.c

Обработка данных и пульсов

40-43 Вызываем функцию `select` и проверяем код возврата.

44-54 Таймаут обрабатывается так же, как в листинге 2.24, только пульсы посылаются через сокет `shb`.

55-62 Если через сокет `shb` пришли данные, читаем их, но ничего не делаем.

63-71 Если данные пришли через сокет `sdata`, читаем столько, сколько сможем, и обрабатываем. Обратите внимание, что теперь производится работа не с сообщениями фиксированной длины. Поэтому читается не больше, чем помещается в буфер. Если данных меньше длины буфера, вызов `recv` вернет все, что есть, но не заблокирует программу. Если

данных больше, то из сокета еще можно читать. Поэтому следующий вызов `select` немедленно вернет управление, и можно будет обработать очередную порцию данных.

72-73 Поскольку только что пришло сообщение от сервера, сбрасываем переменную `heartbeats` в 0 и снова взводим таймер.

И в заключение рассмотрим код сервера для этого примера (листинг 2.28). Как и код клиента, он почти совпадает с исходным сервером (листинг 2.25) за тем исключением, что устанавливает два соединения и работает с двумя сокетами.

Листинг 2.28. Код инициализации и установления соединения на стороне сервера

hb_server2.c

```
#include "etcp.h"
#include "heartbeat.h"

int main( int argc, char **argv )
{
  fd_set allfd;
  fd_set readfd;
  char msg[ 1024 ];
  struct sockaddr_in peer;
  struct timeval tv;
  SOCKET s;
  SOCKET sdata;
  SOCKET shb;
  int rc;
  int maxfd1;
  int missed_heartbeats = 0;
  int peerlen = sizeof( peer );
  char hbmsg[ 1 ];

  INIT();
  s = tcp_server( NULL, argv[ 1 ] );
  sdata = accept( s, ( struct sockaddr * )&peer,
     &peerlen );
  if ( !isvalidsock( sdata ) )
     error( 1, errno, "accept failed" );
  rc = readn( sdata, ( char * )&peer.sin_port,
     sizeof( peer.sin_port ) );
  if ( rc < 0 )
     error( 1, errno, "ошибка при чтении номера порта" );
  shb = socket( PF_INET, SOCK_STREAM, 0 );
  if ( !isvalidsock( shb ) )
     error( 1, errno, "ошибка при создании сокета shb" );
  rc = connect( shb, ( struct sockaddr * )&peer, peerlen );
  if ( rc )
     error( 1, errno, "ошибка вызова connect для сокета shb" );
  tv.tv_sec = T1 + T2;
  tv.tv_usec = 0;
  FD_ZERO( &allfd );
```

```
37    FD_SET( sdata, &allfd );
38    FD_SET( shb, &allfd );
39    maxfd1 = ( sdata > shb ? sdata : shb ) + 1;
```
hb_server2.c

Инициализация и соединение

19-23 Слушаем и принимаем соединения от клиента. Кроме того, сохраняем адрес клиента в переменной `peer`, чтобы знать, с кем устанавливать соединение для пульсации.

24-27 Читаем номер порта, который клиент прослушивает в ожидании соединения для пульсации. Считываем его непосредственно в структуру `peer`. О преобразовании порядка байтов с помощью `htons` или `ntohs` беспокоиться не надо, так как порт уже пришел в сетевом порядке. В таком виде его и надо сохранить в `peer`.

28-33 Получив сокет `shb`, устанавливаем соединение для пульсации.

34-39 Взводим таймер и инициализируем маски для `select`.

Оставшаяся часть сервера представлена в листинге 2.29.

Листинг 2.29. Обработка сообщений сервером

hb_server2.c
```
40 for ( ;; )
41 {
42    readfd = allfd;
43    rc = select( maxfd1, &readfd, NULL, NULL, &tv );
44    if ( rc < 0 )
45       error( 1, errno, "ошибка вызова select" );
46    if ( rc == 0 )  /* Произошел тайм-аут. */
47    {
48       if ( ++missed_heartbeats > 3 )
49          error( 1, 0, "соединения нет\n" );
50       error( 0, 0, "пропущен пульс #%d\n",
51          missed_heartbeats );
52       tv.tv_sec = T2;
53       continue;
54    }
55    if ( FD_ISSET( shb, &readfd ) )
56    {
57       rc = recv( shb, hbmsg, 1, 0 );
58       if ( rc == 0 )
59          error( 1, 0, "клиент завершил работу\n" );
60       if ( rc < 0 )
61          error( 1, errno, "ошибка вызова recv для сокета shb" );
62       rc = send( shb, hbmsg, 1, 0 );
63       if ( rc < 0 )
64          error( 1, errno, "ошибка вызова send для сокета shb" );
65    }
66    if ( FD_ISSET( sdata, &readfd ) )
67    {
```

```
68         rc = recv( sdata, msg, sizeof( msg ), 0 );
69         if ( rc == 0 )
70            error( 1, 0, "клиент завершил работу\n" );
71         if ( rc < 0 )
72            error( 1, errno, "ошибка вызова recv" );
73         /* Обработка данных. */
74      }
75      missed_heartbeats = 0;
76      tv.tv_sec = T1 + T2;
77      }
78      EXIT( 0 );
79 }
```

hb_server2.c

42-45 Как и в ситуации с клиентом, вызываем `select` и проверяем возвращаемое значение.

46-53 Обработка тайм-аута такая же, как и в первом примере сервера в листинге 2.25.

55-65 Если в сокете `shb` есть данные для чтения, то читаем однобайтовый пульс и возвращаем его клиенту.

66-74 Если что-то поступило по соединению для передачи данных, читаем и обрабатываем данные, проверяя ошибки и признак конца файла.

75-76 Поскольку только что получены данные от клиента, соединение все еще живо, поэтому сбрасываем в нуль счетчик пропущенных пульсов и переустанавливаем таймер.

Если запустить клиента и сервер и имитировать сбой в сети, отсоединив один из хостов, то получим те же результаты, что при запуске `hb_server` и `hb_client`.

Резюме

Хотя TCP и не предоставляет средств для немедленного уведомления клиента о потере связи, тем не менее несложно самостоятельно встроить такой механизм в приложение. Здесь рассмотрены две модели реализации контрольных сообщений-пульсов. На первый взгляд, это может показаться избыточным, но одна модель не подходит для всех случаев.

Первый способ применяется, когда приложения обмениваются между собой сообщениями, содержащими поле идентификатора типа. В этом случае все очень просто: достаточно добавить еще один тип для сообщений-пульсов. «Родители» могут спокойно работать – их «дети» под надежным присмотром.

Второй способ применим в ситуации, когда приложения обмениваются потоком байтов без явно выраженных границ сообщений. В качестве примера можно назвать передачу последовательности нажатий клавиш. В данном примере использовано отдельное соединение для приема и передачи пульсов. Разумеется, тот же метод можно было бы применить и в первом случае, но он несколько сложнее, чем простое добавление нового типа сообщения.

В книге «UNIX Network Programming» [Stevens 1998] описан еще один метод организации пульсации с помощью механизма срочных данных, имеющегося

в TCP. Это лишний раз демонстрирует, какие разнообразные возможности имеются в распоряжении прикладного программиста для организации уведомления приложения о потере связи.

Наконец, следует напомнить, что хотя было сказано только о протоколе TCP, то же самое верно и в отношении UDP. Представим сервер, который посылает широковещательные сообщения нескольким клиентам в локальной сети или организует групповое вещание на глобальную сеть. Поскольку соединения нет, клиенты не имеют информации о крахе сервера, хоста или сбое в сети. Если в датаграммах есть поле типа, то серверу нужно лишь определить тип для датаграммы-пульса и посылать ее, когда в сети какое-то время не было других сообщений. Вместо этого он мог бы рассылать широковещательные датаграммы на отдельный порт, который клиенты прослушивают.

Совет 11. Будьте готовы к некорректному поведению партнера

Часто при написании сетевых приложений не учитывают возможность возникновения ошибки, считая ее маловероятной. В связи с этим ниже приведена выдержка из требований к хостам, содержащихся в RFC 1122 [Braden 1989, стр. 12]: «Программа должна обрабатывать любую возможную ошибку, как бы маловероятна она ни была; рано или поздно придет пакет именно с такой комбинацией ошибок и атрибутов, и если программа не готова к этому, то неминуем хаос. Правильнее всего предположить, что сеть насыщена злонамеренными агентами, которые посылают пакеты, специально подобранные так, чтобы вызвать максимально разрушительный эффект. Необходимо думать о том, как защититься, хотя надо признать, что наиболее серьезные проблемы в сети Internet были вызваны непредвиденными механизмами, сработавшими в результате сочетания крайне маловероятных событий. Никакой злоумышленник не додумался бы до такого!»

Сегодня этот совет более актуален, чем во время написания. Появилось множество реализаций TCP, и некоторые из них содержат грубые ошибки. К тому же все больше программистов разрабатывают сетевые приложения, но далеко не у всех есть опыт работы в этой области.

Однако самый серьезный фактор – это лавинообразный рост числа подключенных к Internet персональных компьютеров. Ранее можно было предполагать, что у пользователей есть хотя бы минимальная техническая подготовка, они понимали, к каким последствиям приведет, скажем, выключение компьютера без предварительного завершения сетевого приложения. Теперь это не так.

Поэтому особенно важно практиковать защитное программирование и предвидеть все действия, которые может предпринять хост на другом конце, какими бы маловероятными они ни казались. Эта тема уже затрагивалась в совете 9 при обсуждении потенциальных ошибок при работе с TCP, а также в совете 10, где речь шла об обнаружении потери связи. В этом разделе будет рассмотрено, какие действия вашего партнера могут нанести ущерб. Главное – не думайте, что он будет следовать прикладному протоколу, даже если обе стороны протокола реализовывали вы сами.

Проверка завершения работы клиента

Предположим, что клиент извещает о желании завершить работу, посылая серверу запрос из одной строки, в которой есть только слово quit. Допустим далее, что сервер читает строки из входного потока с помощью функции readline (ее текст приведен в листинге 2.32), которая была описана в совете 9. Что произойдет, если клиент завершится (аварийно или нормально) раньше, чем пошлет команду quit? TCP на стороне клиента отправит сегмент FIN, после чего операция чтения на сервере вернет признак конца файла. Конечно, это просто обнаружить, только сервер должен обязательно это сделать. Легко представить себе такой код, предполагая правильное поведение клиента:

```
for ( ;; )
{
   if ( readline( s, buf, sizeof( buf ) ) < 0 )
      error( 1, errno, "ошибка вызова readline" );
   if ( strcmp( buf, "quit\n" ) == 0)
      /* Выполнить функцию завершения клиента. */
   else
      /* Обработать запрос. */
}
```

Хотя код выглядит правильным, он не работает, поскольку будет повторно обрабатывать последний запрос, если клиент завершился, не послав команду quit.

Предположим, что вы увидели ошибку в предыдущем фрагменте (или нашли ее после долгих часов отладки) и изменили код так, чтобы явно обнаруживался признак конца файла:

```
for ( ;; )
{
   rc = readline( s, buf, sizeof( buf ) );
   if ( rc < 0 )
      error( 1, errno, "ошибка вызова readline" );
   if ( rc == 0 || strcmp( buf, "quit\n" ) == 0)
      /* Выполнить функцию завершения клиента. */
   else
      /* Обработать запрос. */
}
```

И этот код тоже неправилен, так как в нем не учитывается случай, когда хост клиента «падает» до того, как клиент послал команду quit или завершил работу. В этом месте легко принять неверное решение, даже осознавая проблему. Для проверки краха клиентского хоста надо ассоциировать таймер с вызовом readline. Потребуется примерно в два раза больше кода, если нужно организовать обработку «безвременной кончины» клиента. Представив себе, сколько придется писать, вы решаете, что шансов «грохнуться» хосту клиента мало.

Но проблема в том, что хосту клиента и необязательно завершаться. Если это ПК, то пользователю достаточно выключить его, не выйдя из программы. А это очень легко, поскольку клиент мог исполняться в свернутом окне или в окне,

закрытом другими, так что пользователь про него, вероятно, забыл. Есть и другие возможности. Если соединение между хостами установлено с помощью модема на клиентском конце (так сегодня выполняется большинство подключений к Internet), то пользователь может просто выключить модем. Шум в линии также может привести к обрыву соединения. И все это с точки зрения сервера неотличимо от краха хоста клиента.

Примечание *При некоторых обстоятельствах ошибку, связанную с модемом, можно исправить, повторно набрав номер (помните, что TCP способен восстанавливаться после временных сбоев в сети), но зачастую IP-адреса обоих оконечных абонентов назначаются динамически сервис-провайдером при установлении соединения. В таком случае маловероятно, что будет задан тот же адрес, и поэтому клиент не сможет оживить соединение.*

Для обнаружения потери связи с клиентом необязательно реализовывать пульсацию, как это делалось в совете 10. Нужно всего лишь установить тайм-аут для операции чтения. Тогда, если от клиента в течение определенного времени не поступает запросов, то сервер может предположить, что клиента больше нет, и разорвать соединение. Так поступают многие FTP-серверы. Это легко сделать, либо явно установив таймер, либо воспользовавшись возможностями системного вызова `select`, как было сделано при реализации пульсации.

Если вы хотите, чтобы сервер не «зависал» навечно, то можете воспользоваться механизмом контролеров для разрыва соединения по истечении контрольного тайм-аута. В листинге 2.30 приведен простой TCP-сервер, который принимает сообщение от клиента, читает из сокета и пишет результат на стандартный вывод. Чтобы сервер не «завис», следует задать для сокета опцию `SO_KEEPALIVE` с помощью вызова `setsockopt`. Четвертый аргумент `setsockopt` должен указывать на ненулевое целое число, если надо активировать посылку контролеров, или на нулевое целое, чтобы ее отменить.

Запустите этот сервер на машине `bsd`, а на другой машине – программу telnet в качестве клиента. Соединитесь с сервером, отправьте ему строку «`hello`», чтобы соединение точно установилось, а затем отключите клиентскую систему от сети. Сервер напечатает следующее:

```
bsd: $ keep 9000
hello
                                          Клиент отключился от сети.
                                          ...
                                          Спустя 2 ч 11 мин 15 с.
keep: ошибка вызова recv: Operation timed out (60)
bsd: $
```

Как и следовало ожидать, TCP на машине `bsd` разорвал соединение и вернул серверу код ошибки `ETIMEDOUT`. В этот момент сервер завершает работу и освобождает все ресурсы.

Листинг 2.30. Сервер, использующий механизм контролеров

keep.c

```
#include "etcp.h"
int main( int argc, char **argv )
{
  SOCKET s;
  SOCKET s1;
  int on = 1;
  int rc;
  char buf[ 128 ];

  INIT();
  s = tcp_server( NULL, argv[ 1 ] );
  s1 = accept( s, NULL, NULL );
  if ( !isvalidsock( s1 ) )
     error( 1, errno, "ошибка вызова accept\n" );
  if ( setsockopt( s1, SOL_SOCKET, SO_KEEPALIVE,
     ( char * )&on, sizeof( on ) ) )
     error( 1, errno, "ошибка вызова setsockopt" );
  for ( ;; )
  {
     rc = readline( s1, buf, sizeof( buf ) );
     if ( rc == 0 )
        error( 1, 0, "другой конец отключился\n" );
     if ( rc < 0 )
        error( 1, errno, "ошибка вызова recv" );
     write( 1, buf, rc );
     }
}
```

keep.c

Проверка корректности входной информации

Что бы вы ни программировали, не думайте, что приложение будет получать только те данные, на которые рассчитывает. Пренебрежение этим принципом – пример отсутствия защитного программирования. Хочется надеяться, что профессиональный программист, разрабатывающий коммерческую программу, всегда ему следует. Однако часто это правило игнорируют. В работе [Miller et al. 1995] описывается, как авторы генерировали случайный набор входных данных и подавали его на вход всевозможных стандартных утилит UNIX от разных производителей. При этом им удалось «сломать» (с дампом памяти) или «подвесить» (в бесконечном цикле) от 6 до 43% тестируемых программ (в зависимости от производителя). В семи исследованных коммерческих системах частота отказов составила 23%.

Вывод ясен: если такие результаты получены при тестировании зрелых программ, которые принято считать программами «промышленного качества», то тем более необходимо защищаться и подвергать сомнению все места в программе, где неожиданные входные данные могут привести к нежелательным результатам. Рассмотрим несколько примеров, когда неожиданные данные оказываются источником ошибок.

Две самые распространенные причины краха приложений – это переполнение буфера и сбитые указатели. В вышеупомянутом исследовании именно эти две ошибки послужили причиной большинства сбоев. Можно сказать, что в сетевых программах переполнение буфера должно быть редким явлением, так как при обращении к системным вызовам, выполняющим чтение (`read`, `recv`, `recvfrom`, `readv` и `readmsg`), всегда необходимо указывать размер буфера. Но вы увидите далее, как легко допустить такую ошибку. (Это рассмотрено в замечании к строке 42 программы `shutdown.c` в совете 16.)

Чтобы понять, как это происходит, разработаем функцию `readline`, использовавшуюся в совете 9. Поставленная задача – написать функцию, которая считывает из сокета в буфер одну строку, заканчивающуюся символом новой строки, и дописывает в конец двоичный нуль. На начало буфера указывает параметр `buf`.

```
#include "etcp.h"

int readline( SOCKET s, char *buf, size_t len );
```

Возвращаемое значение: число прочитанных байтов или –1 в случае ошибки.

Первая попытка реализации, которую надо отбросить сразу, похожа на следующий код:

```
while ( recv( fd, , &c, 1, 0 ) == 1 )
{
   *bufptr++ = c;
   if ( c == "\n" )
      break;
}
/* Проверка ошибок, добавление завершающего нуля и т.д. */
```

Прежде всего, многократные вызовы `recv` совсем неэффективны, поскольку при каждом вызове нужно два переключения – в режим ядра и обратно.

Примечание *Но иногда приходится писать и такой код – смотрите, например, функцию `readcrlf` в листинге 3.10.*

Важнее, однако, то, что нет контроля переполнения буфера.

Чтобы понять, как аналогичная ошибка может вкрасться и в более рациональную реализацию, следует рассмотреть такой фрагмент:

```
static char *bp;
static int cnt = 0;
static char b[ 1500 ];
char c;
for ( ;; )
{
   if (cnt-- <= 0)
   {
      cnt = recv( fd, b, sizeof( b ), 0 );
      if ( cnt < 0 )
```

```
            return -1;
        if ( cnt == 0 )
            return 0;
        bp = b;
    }
    c = *bp++;
    *bufptr++ = c;
    if ( c == "\n")
    {
        *bufptr = "\0";
        break;
    }
}
```

В этой реализации нет неэффективности первого решения. Теперь считывается большой блок данных в промежуточный буфер, а затем по одному копируются байты в окончательный буфер; по ходу производится поиск символа новой строки. Но при этом в коде присутствует та же ошибка, что и раньше. Не проверяется переполнение буфера, на который указывает переменная `bufptr`. Можно было бы и не писать универсальную функцию чтения строки; такой код – вместе с ошибкой – легко мог бы быть частью какой-то большей функции.

А теперь напишем настоящую реализацию (листинг 2.31).

Листинг 2.31. Неправильная реализация readline

readline.c

```
int readline( SOCKET fd, char *bufptr, size_t len )
{
    char *bufx = bufptr;
    static char *bp;
    static int cnt = 0;
    static char b[ 1500 ];
    char c;

    while ( len -- > 0 )
    {
        if ( cnt -- <= 0 )
        {
            cnt = recv( fd, b, sizeof( b ), 0 );
            if ( cnt < 0 )
                return -1;
            if ( cnt == 0 )
                return 0;
            bp = b;
        }
        c = *bp++;
        *bufptr++ = c;
        if ( c == "\n" )
        {
            *bufptr = "\0";
```

```
24            return bufptr - bufx;
25        }
26    }
27    set_errno( EMSGSIZE );
28    return -1;
29 }
```

readline.c

На первый взгляд, все хорошо. Размер буфера передается `readline` и во внешнем цикле проверяется, не превышен ли он. Если размер превышен, то переменной `errno` присваивается значение `EMSGSIZE` и возвращается –1.

Чтобы понять, в чем ошибка, представьте, что функция вызывается так:

```
rc = readline( s, buffer, 10 );
```

и при этом из сокета читается строка

```
123456789<nl>
```

Когда в `c` записывается символ новой строки, значение `len` равно нулю. Это означает, что данный байт последний из тех, что готовы принять. В строке 20 помещаете символ новой строки в буфер и продвигаете указатель `bufptr` за конец буфера. Ошибка возникает в строке 23, где записывается нулевой байт за границу буфера.

Заметим, что похожая ошибка имеет место и во внутреннем цикле. Чтобы увидеть ее, представьте, что при входе в функцию `readline` значение `cnt` равно нулю и `recv` возвращает один байт. Что происходит дальше? Можно назвать это «опустошением» (underflow) буфера.

Этот пример показывает, как легко допустить ошибки, связанные с переполнением буфера, даже предполагая, что все контролируется. В листинге 2.32 приведена окончательная, правильная версия `readline`.

Листинг 2.32. Окончательная версия readline

readline.c

```
int readline( SOCKET fd, char *bufptr, size_t len )
{
char *bufx = bufptr;
static char *bp;
static int cnt = 0;
static char b[ 1500 ];
char c;

while ( --len > 0 )
{
  if ( --cnt <= 0 )
  {
     cnt = recv( fd, b, sizeof( b ), 0 );
     if ( cnt < 0 )
     {
        if ( errno == EINTR )
        {
```

```
            len++; /* Уменьшим на 1 в заголовке while. */
            continue;
         }
         return -1;
      }
      if ( cnt == 0 )
         return 0;
      bp = b;
   }
   c = *bp++;
   *bufptr++ = c;
   if ( c == "\n" )
   {
      *bufptr = "\0";
      return bufptr - bufx;
   }
   }
   set_errno( EMSGSIZE );
   return -1;
}
```
readline.c

Единственная разница между этой и предыдущей версиями в том, что уменьшаются значения `len` и `cnt` до проверки, а не после. Также проверяется, не вернула ли `recv` значение `EINTR`. Если это так, то вызов следует повторить. При уменьшении `len` до использования появляется гарантия, что для нулевого байта всегда останется место. А, уменьшая `cnt`, можно получить некоторую уверенность, что данные не будут читаться из пустого буфера.

Резюме

Вы всегда должны быть готовы к неожиданным действиям со стороны пользователей и хостов на другом конце соединения. В этом разделе рассмотрено два примера некорректного поведения другой стороны. Во-первых, нельзя надеяться на то, что партнер обязательно сообщит вам о прекращении передачи данных. Во-вторых, продемонстрирована важность проверки правильности входных данных и разработана функция `readline`, устойчивая к ошибкам.

Совет 12. Не думайте, что программа, работающая в локальной сети, будет работать и в глобальной

Многие сетевые приложения разрабатываются и тестируются в локальной сети или даже на одной машине. Это просто, удобно и недорого, но при этом могут остаться незамеченными некоторые ошибки.

Несмотря на возможную потерю данных, показанную в совете 7, локальная сеть представляет собой среду, в которой датаграммы почти никогда не теряются, не задерживаются и практически всегда доставляются в правильном порядке. Однако из этого не следует делать вывод, что приложение, замечательно работающее

в локальной сети, будет также хорошо функционировать и в глобальной сети или в Internet. Здесь можно столкнуться с проблемами двух типов:

- производительность глобальной сети оказывается недостаточной из-за дополнительных сетевых задержек;
- некорректный код, работавший в локальной сети, отказывает в глобальной.

Если вам встречается проблема первого типа, то, скорее всего, приложение следует перепроектировать.

Недостаточная производительность

Чтобы получить представление о такого рода проблемах, изменим программы `hb_server` (листинг 2.25) и `hb_client` (листинг 2.24), задав T1, равным 2 с, а T2 – 1 с (листинг 2.23). Тогда пульс будет посылаться каждые две секунды, и при отсутствии ответа в течение трех секунд приложение завершится.

Сначала запустим эти программы в локальной сети. Проработав почти семь часов, сервер сообщил о пропуске одного пульса 36 раз, а о пропуске двух пульсов – один раз. Клиенту пришлось посылать второй пульс 11 из 12139 раз. И клиент, и сервер работали, пока клиент не остановили вручную. Такие результаты типичны для локальной сети. Если не считать редких и небольших задержек, сообщения доставляются своевременно.

А теперь запустим те же программы в Internet. Спустя всего лишь 12 мин клиент сообщает, что послал три пульса, не получив ответа, и завершает сеанс. Распечатка выходной информации от клиента, частично представленная ниже, показывает, как развивались события:

```
sparc: $ hb_client 205.184.151.171 9000
hb_client: посылаю пульс: #1
hb_client: посылаю пульс: #2
hb_client: посылаю пульс: #3
hb_client: посылаю пульс: #1
hb_client: посылаю пульс: #2
hb_client: посылаю пульс: #1
                                           Много строк опущено.
hb_client: посылаю пульс: #1
hb_client: посылаю пульс: #2
hb_client: посылаю пульс: #1
hb_client: посылаю пульс: #2
hb_client: посылаю пульс: #3
hb_client: посылаю пульс: #1
hb_client: посылаю пульс: #2
hb_client:                                 Соединение завершается через
                                           1 с после последнего пульса.
sparc: $
```

В этот раз клиент послал первый пульс 251 раз, а второй – 247 раз. Таким образом, он почти ни разу не получил вовремя ответ на первый пульс. Десять раз клиенту пришлось посылать третий пульс.

Сервер также продемонстрировал значительное падение производительности. Тайм-аут при ожидании первого пульса происходил 247 раз, при ожидании второго пульса – 5 и при ожидании третьего пульса – 1 раз.

Этот пример показывает, что приложение, которое прекрасно работает в условиях локальной сети, может заметно снизить производительность в глобальной.

Скрытая ошибка

В качестве примера проблемы второго типа рассмотрим основанное на TCP приложение, занимающееся телеметрией. Здесь сервер каждую секунду принимает от удаленного датчика пакет с результатами измерений. Пакет может состоять из двух или трех целочисленных значений. В примитивной реализации подобного сервера мог бы присутствовать такой цикл:

```
int pkt[ 3 ];
for ( ;; )
{
  rc = recv( s, ( char * ) pkt, sizeof( pkt ), 0 );
  if ( rc != sizeof( int ) * 2 && rc != sizeof( int ) * 3 )
     /* Протоколировать ошибку и выйти. */
  else
     /* Обработать rc / sizeof( int ) значений. */
}
```

Из совета 6 вы знаете, что этот код некорректен, но попробуем провести простое моделирование. Напишем сервер (листинг 2.33), в котором реализован только что показанный цикл.

Листинг 2.33. Моделирование сервера телеметрии

telemetrys.c

```
#include "etcp.h"
#define TWOINTS      ( sizeof( int ) * 2 )
#define THREEINTS    ( sizeof( int ) * 3 )
int main( int argc, char **argv )
{
  SOCKET s;
  SOCKET s1;
  int rc;
  int i = 1;
  int pkt[ 3 ];

  INIT();
  s = tcp_server( NULL, argv[ 1 ] );
  s1 = accept( s, NULL, NULL );
  if ( !isvalidsock( s1 ) )
     error( 1, errno, "ошибка вызова accept" );
  for ( ;; )
  {
     rc = recv( s1, ( char * )pkt, sizeof( pkt ), 0 );
     if ( rc != TWOINTS && rc != THREEINTS )
```

```
20          error( 1, 0, "recv вернула %d\n", rc );
21       printf( "Пакет %d содержит %d значений в %d байтах\n",
22          i++, ntohl( pkt[ 0 ] ), rc );
23    }
24 }
```
telemetrys.c

11-15 В этих строках реализована стандартная инициализация и прием соединения.

16-23 В данном цикле принимаются данные от клиента. Если получено при чтении не в точности `sizeof( int ) * 2` или `sizeof( int ) * 3` байт, то протоколируем ошибку и выходим. В противном случае байты первого числа преобразуются в машинный порядок (совет 28), а затем результат и число прочитанных байтов печатаются на `stdout`. В листинге 2.34 вы увидите, что клиент помещает число значений в первое число, посылаемое в пакете. Это поможет разобраться в том, что происходит. Здесь не используется это число как «заголовок сообщения», содержащий его размер (совет 6).

Для тестирования этого сервера также необходим клиент, который каждую секунду посылает пакет целых чисел, имитируя работу удаленного датчика. Текст клиента приведен в листинге 2.34.

Листинг 2.34. Имитация клиента для сервера телеметрии

telemetryc.c

```
 1 #include "etcp.h"
 2 int main( int argc, char **argv )
 3 {
 4    SOCKET s;
 5    int rc;
 6    int i;
 7    int pkt[ 3 ];
 8    INIT();
 9    s = tcp_client( argv[ 1 ], argv[ 2 ] );
10    for ( i = 2;; i = 5 - i )
11    {
12       pkt[ 0 ] = htonl( i );
13       rc = send( s, ( char * )pkt, i * sizeof( int ), 0 );
14       if ( rc < 0 )
15          error( 1, errno, "ошибка вызова send" );
16       sleep( 1 );
17    }
18 }
```
telemetryc.c

8-9 Производим инициализацию и соединяемся с сервером.

10-17 Каждую секунду посылаем пакет из двух или трех целых чисел. Как говорилось выше, первое число в пакете – это количество последующих чисел (преобразованное в сетевой порядок байтов).

Для тестирования модели запустим сервер на машине bsd, а клиента – на машине sparc. Сервер печатает следующее:

```
bsd: $ telemetrys 9000
Пакет 1 содержит 2 значения в 8 байтах
Пакет 2 содержит 3 значения в 12 байтах
                                        Много строк опущено.
Пакет 22104 содержит 3 значения в 12 байтах
Пакет 22105 содержит 2 значения в 8 байтах
                                        Клиент завершил сеанс через
                                        6 ч 8 мин 15 с.
telemetrys: recv вернула 0
bsd: $
```

Хотя в коде сервера есть очевидная ошибка, он проработал в локальной сети без сбоев более шести часов, после чего моделирование завершили с помощью ручной остановки клиента.

Примечание *Протокол сервера проверен с помощью сценария, написанного на awk – необходимо убедиться, что каждая операция чтения вернула правильное число байтов.*

Однако при запуске того же сервера через Internet результаты получились совсем другие. Опять запустим клиента на машине sparc, а сервер – на машине bsd, но на этот раз заставим клиента передавать данные через глобальную сеть, указав ему адрес сетевого интерфейса, подключенного к Internet. Как видно из последних строк, напечатанных сервером, фатальная ошибка произошла уже через 15 мин.

```
Пакет 893 содержит 2 значения в 8 байтах
Пакет 894 содержит 3 значения в 12 байтах
Пакет 895 содержит 2 значения в 12 байтах
Пакет 896 содержит -268436204 значения в 8 байтах
Пакет 897 содержит 2 значения в 12 байтах
Пакет 898 содержит -268436204 значения в 8 байтах
Пакет 899 содержит 2 значения в 12 байтах
Пакет 900 содержит -268436204 значения в 12 байтах
telemetrys: recv вернула 4
bsd: $
```

Ошибка произошла при обработке пакета 895, когда нужно было прочесть 8 байт, а прочли 12. На рис. 2.21 представлено, что произошло.

Числа слева показывают, сколько байтов было в приемном буфере TCP на стороне сервера. Числа справа – сколько байтов сервер реально прочитал. Вы видите, что пакеты 893 и 894 доставлены и обработаны, как и ожидалось. Но, когда telemetrys вызвал recv для чтения пакета 895, в буфере было 20 байт.

Примечание *Трассировка сетевого трафика, полученная с помощью программы* tcpdump *(совет 34), показывает, что в этот момент были потеряны TCP-сегменты, которыми обменивались два хоста.*

Вероятно, причиной послужила временная перегрузка сети, из-за которой промежуточный маршрутизатор отбросил пакет. Перед доставкой пакета 895 клиент `telemetryc` уже подготовил пакет 896, и оба были доставлены вместе.

В пакете 895 было 8 байт, но, поскольку уже пришел пакет 896, сервер прочитал пакет 895 и первое число из пакета 896. Поэтому в распечатке видно, что было прочитано 12 байт, хотя пакет 895 содержит только два целых. При следующем чтении возвращено два целых из пакета 896, и `telemetrys` напечатал мусор вместо числа значений, так как `telemetryc` не инициализировал второе значение.

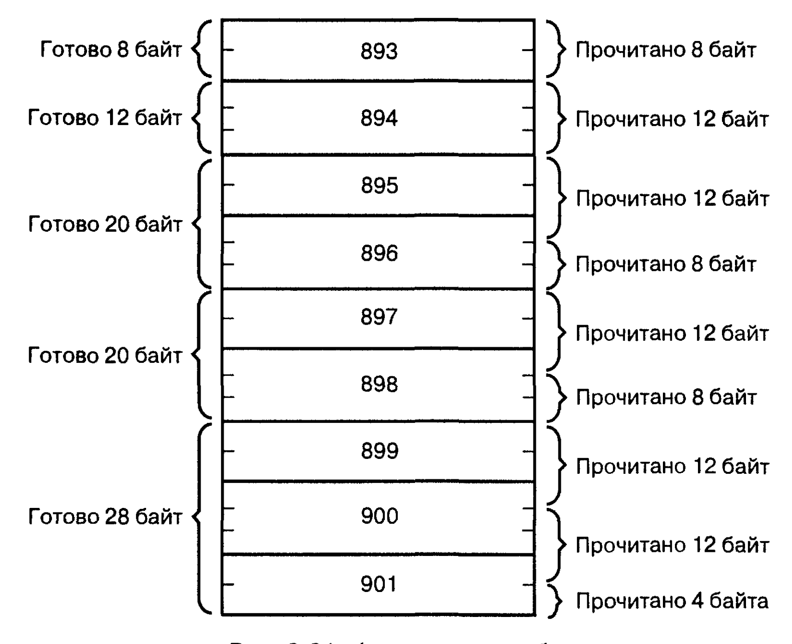


Рис. 2.21. Фатальная ошибка

Как видно из рис. 2.21, то же самое произошло с пакетами 897 и 898, так что при следующем чтении было доступно уже 28 байт. Теперь `telemetrys` читает пакет 899 и первое значение из пакета 900, остаток пакета 900 и первое значение из пакета 901 и наконец последнее значение из пакета 901. Последняя операция чтения возвращает только 4 байта, поэтому проверка в строке 19 завершается неудачно, а моделирование – с ошибкой.

К сожалению, на более раннем этапе моделирования произошло еще худшее.

```
Пакет 31 содержит 2 значения в 8 байтах
Пакет 32 содержит 3 значения в 12 байтах
Пакет 33 содержит 2 значения в 12 байтах
Пакет 34 содержит -268436204 значения в 8 байтах
Пакет 35 содержит 2 значения в 8 байтах
Пакет 36 содержит 3 значения в 12 байтах
```

Всего через 33 с после начала моделирования произошла ошибка, оставшаяся необнаруженной. Как показано на рис. 2.22, когда `telemetrys` читал пакет 33, в буфере было 20 байт, поэтому операция чтения вернула 12 байт вместо 8. Это означает, что пакет с двумя значениями ошибочно был принят за пакет с тремя значениями, а затем наоборот. Начиная с пакета 35, `telemetrys` восстановил синхронизацию, и ошибка прошла незамеченной.

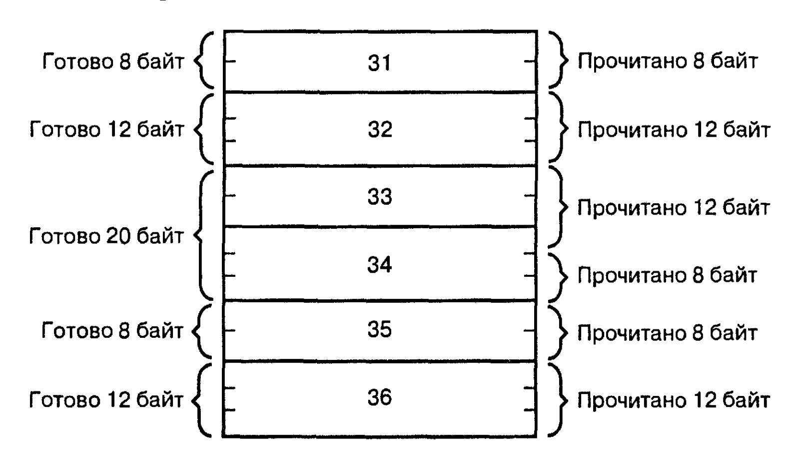


Рис. 2.22. Незамеченная ошибка

Резюме

Локальная сеть, которая представляет собой почти идеальную среду, может маскировать проблемы производительности и даже ошибки. Не думайте, что приложение, работающее в локальной сети, будет также хорошо работать и в глобальной.

Из-за сетевых задержек приложение, производительность которого в локальной сети была удовлетворительной, в глобальной сети может работать неприемлемо медленно. В результате иногда приходится перепроектировать программу.

Из-за перегрузок в интенсивно используемой глобальной сети, особенно в Internet, данные могут доставляться как внезапно, так и пакетами неожиданного размера. Это требует от вас особой осторожности в допущениях о том, сколько данных может прийти в определенный момент и с какой частотой они поступают.

Хотя в этом разделе говорилось исключительно о протоколе TCP, то же относится и к UDP, поскольку он не обладает встроенной надежностью, чтобы противостоять тяжелым условиям в Internet.

Совет 13. Изучайте работу протоколов

В книге [Stevens 1998] автор отмечает, что основные проблемы в сетевом программировании не имеют отношения ни к программированию, ни к API. Они возникают из-за непонимания работы сетевых протоколов. Это подтверждают вопросы, которые задают в конференциях, посвященных сетям (совет 44). Например, некто, читая справочную документацию на своей UNIX- или Windows-машине, обнаруживает, как отключить алгоритм Нейгла. Но если он не понимает принципов

управления потоком, заложенных в TCP, и роли этого алгоритма, то вряд ли разберется, когда имеет смысл его отключать, а когда – нет.

Точно так же, отсутствие механизма немедленного уведомления о потере связи, обсуждавшееся в совете 10, может показаться серьезным недостатком, если вы не понимаете, почему было принято такое решение. Разобравшись с причинами, можно без труда организовать обмен сообщениями-пульсами именно с той частотой, которая нужна конкретному приложению.

Есть несколько способов изучить протоколы, и многие из них будут рассмотрены в главе 4. Основной источник информации о протоколах TCP/IP – это RFC, который официально определяет, как они *должны* работать. В RFC обсуждается широкий спектр вопросов разной степени важности, в том числе все протоколы из семейства TCP/IP. Все RFC, а также сводный указатель находятся на следующем сайте: http://www.rfc-editor.org.

В совете 43 описаны также другие способы получения RFC.

Поскольку RFC – это плод труда многих авторов, они сильно различаются доступностью изложения. Кроме того, некоторые вопросы освещаются в нескольких RFC и не всегда просто составить целостную картину.

Существуют и другие источники информации о протоколах, более понятные для начинающих. Два из них будут рассмотрены здесь, а остальные – в главе 4.

В книге [Comer 1995] описываются основные протоколы TCP/IP и то, как они должны работать, с точки зрения RFC. Здесь содержатся многочисленные ссылки на RFC, которые облегчают дальнейшее изучение предмета и дают общее представление об организации RFC. Поэтому некоторые считают эту книгу теоретическим введением в противоположность книгам [Stevens 1994; Stevens 1995], где представлен подход, ориентированный в основном на практическое применение.

В книгах Стивенса семейство протоколов TCP/IP исследуется с точки зрения реализации. Иными словами, показывается, как основные реализации TCP/IP работают в действительности. В качестве инструмента исследования используются, главным образом, данные, выдаваемые программой `tcpdump` (совет 34), и временные диаграммы типа изображенной на рис. 2.16. В сочетании с детальным изложением форматов пакетов и небольшими тестовыми программами, призванными прояснить некоторые аспекты работы обсуждаемых протоколов, это дает возможность ясно представить себе их функционирование. С помощью формального описания добиться этого было бы трудно.

Хотя в этих книгах приняты разные подходы к освещению протоколов TCP/IP, не следует думать, будто один подход чем-то лучше другого, а стало быть, отдавать предпочтение только одной книге. Полезность каждой книги зависит от поставленной перед вами задачи в данный момент. По сути, издания взаимно дополняют друг друга. И серьезный программист, занимающийся разработкой сетевых приложений, должен включить эти две книги в свою библиотеку.

Резюме

В этом разделе обсуждалось, насколько важно разбираться в функционировании протоколов. Отмечено, что официальной спецификацией TCP/IP являются RFC и рекомендованы книги Комера и Стивенса в качестве дополнительного источника информации о протоколах и их работе.

Совет 14. Не воспринимайте слишком серьезно семиуровневую эталонную модель OSI

Поскольку задача проектирования и реализации сетевых протоколов очень сложна, обычно ее разделяют на меньшие части, более простые для понимания. Традиционно для этого используется концепция уровней. Каждый уровень предоставляет сервисы уровням выше себя и пользуется сервисами нижележащих уровней.

Например, на рис. 2.1, где изображен упрощенный стек протоколов TCP/IP, уровень IP предоставляет сервис, именуемый доставкой датаграмм, уровням TCP и UDP. Чтобы обеспечить такой сервис, IP пользуется сервисами для передачи датаграмм физическому носителю, которые предоставляет уровень сетевого интерфейса.

Модель OSI

Наверное, самый известный пример многоуровневой схемы сетевых протоколов – это эталонная модель открытого взаимодействия систем (Reference Model of Open Systems Interconnection), предложенная Международной организацией по стандартизации (ISO).

Примечание *Многие ошибочно полагают, что в модели OSI были впервые введены концепции разбиения на уровни, виртуализации и многие другие. На самом деле, эти идеи были хорошо известны и активно применялись разработчиками сети ARPANET, которые создали семейство протоколов TCP/IP задолго до появления модели OSI. Об истории этого вопроса вы можете узнать в RFC 871 [Padlipsky 1982].*

7	Прикладной уровень
6	Уровень представления
5	Сеансовый уровень
4	Транспортный уровень
3	Сетевой уровень
2	Канальный уровень
1	Физический уровень

Рис. 2.23. Семиуровневая талонная модель OSI

Поскольку в этой модели семь уровней (рис. 2.23), ее часто называют семиуровневой моделью OSI.

Как уже отмечалось, уровень N предоставляет сервисы уровню N+1 и пользуется сервисами, предоставляемыми уровнем N–1. Кроме того, каждый уровень может взаимодействовать только со своими непосредственными соседями сверху и снизу. Это взаимодействие происходит посредством четко определенных интерфейсов между соседними уровнями, поэтому в принципе реализацию любого уровня можно заменить при условии, что новая реализация предоставляет в точности те же сервисы, и это не отразится на остальных уровнях. Одноименные уровни в коммуникационных стеках обмениваются данными (по сети) с помощью протоколов.

Эти уровни часто упоминаются в литературе по вычислительным сетям. Каждый из них предоставляет следующие сервисы:

- *физический уровень.* Этот уровень связан с аппаратурой. Здесь определяются электрические и временные характеристики интерфейса, способ передачи битов физическому носителю, кадрирование и даже размеры и форма разъемов;

- *канальный уровень.* Это программный интерфейс к физическому уровню. В его задачу входит предоставление надежной связи с последним. На этом уровне находятся драйверы устройств, используемые сетевым уровнем для общения с физическими устройствами. Кроме того, этот уровень обеспечивает формирование кадров для канала, проверку контрольных сумм с целью обнаружения искажения данных и управление совместным доступом к физическому носителю. Обычно задача интерфейса между сетевым и канальным уровнями – создание механизма, обеспечивающего независимость от конкретного устройства;
- *сетевой уровень.* Этот уровень занимается доставкой пакетов от одного узла другому. Он отвечает за адресацию и маршрутизацию, фрагментацию и сборку, а также за управление потоком и предотвращение перегрузок;
- *транспортный уровень.* Реализует надежную сквозную связь между узлами сети, а также управление потоком и предотвращение перегрузок. Он компенсирует ненадежность, присущую нижним уровням, за счет обработки ошибок, которые вызваны искажением данных, потерей пакетов и их доставкой не по порядку;
- *сеансовый уровень.* Транспортный уровень предоставляет надежный полнодуплексный коммуникационный канал между двумя узлами. Сеансовый уровень добавляет такие сервисы, как организация и уничтожение сеанса (например, вход в систему и выход из нее), управление диалогом (эмуляция полудуплексного терминала), синхронизация (создание контрольных точек при передаче больших файлов) и иные надстройки над надежным протоколом четвертого уровня;
- *уровень представления.* Отвечает за представление данных, например, преобразование форматов (скажем, из кода ASCII в код EBCDIC) и сжатие;
- *прикладной уровень.* На нем располагаются пользовательские программы, использующиеся остальными четырьмя уровнями для обмена данными. Известные из мира TCP/IP примеры – это telnet, ftp, почтовые клиенты и Web-браузеры.

Официальное описание семиуровневой модели OSI приведено в документе [International Standards Organization 1984], но оно лишь в общих чертах декларирует, что должен делать каждый уровень. Детальное описание сервисов, предоставляемых протоколами на отдельных уровнях, содержится в других документах ISO. Довольно подробное объяснение модели и ее различных уровней со ссылками на соответствующие документы ISO можно найти в работе [Jain and Agrawala 1993].

Хотя модель OSI полезна как основа для обсуждения сетевых архитектур и реализаций, ее нельзя рассматривать как готовый чертеж для создания любой сетевой архитектуры. Не следует также думать, что размещение некоторой функции на уровне N в этой модели означает, что только здесь наилучшее для нее место.

Модель OSI имеет множество недостатков. Хотя, в конечном итоге, были созданы работающие реализации, протоколы OSI на сегодняшний день утратили актуальность. Основные проблемы этой модели в том, что, во-первых, распределение функций между уровнями произвольно и не всегда очевидно, во-вторых, она была

спроектирована (комитетом) без готовой реализации. Вспомните, как разрабатывался TCP/IP, стандарты которого основаны на результатах экспериментов.

Другая проблема модели OSI – это сложность и неэффективность. Некоторые функции выполняются сразу на нескольких уровнях. Так, обнаружение и исправление ошибок происходит на большинстве уровней.

Как отмечено в книге [Tanenbaum 1996], один из основных дефектов модели OSI состоит в том, что она страдает «коммуникационной ментальностью». Это относится и к терминологии, отличающейся от общеупотребительной, и к спецификации примитивов интерфейсов между уровнями, которые более пригодны для телефонных, а не вычислительных сетей.

Наконец, выбор именно семи уровней продиктован, скорее, политическими, а не техническими причинами. В действительности сеансовый уровень и уровень представления редко встречаются в реально работающих сетях.

Модель TCP/IP

Сравним модель OSI с моделью TCP/IP. Важно отдавать себе отчет в том, что модель TCP/IP документирует дизайн семейства протоколов TCP/IP. Ее не предполагалось представлять в качестве эталона, как модель OSI. Поэтому никто и не рассматривает ее как основу для проектирования новых сетевых архитектур. Тем не менее поучительно сравнить две модели и посмотреть, как уровни TCP/IP отображаются на уровни модели OSI. По крайней мере, это напоминает, что модель OSI – не единственный правильный путь.



Рис. 2.24. Сравнение модели OSI и стека TCP/IP

Как видно из рис. 2.24, стек протоколов TCP/IP состоит из четырех уровней. На прикладном уровне решаются все задачи, свойственные прикладному уровню, уровню представления и сеансовому уровню модели OSI. Транспортный уровень аналогичен соответствующему уровню в OSI и занимается сквозной доставкой. На транспортном уровне определены протоколы TCP и UDP, на межсетевом – протоколы IP, ICMP и IGMP (Internet Group Management Protocol). Он соответствует сетевому уровню модели OSI.

Примечание *С протоколом IP вы уже знакомы. ICMP (Internet Control Message Protocol) – это межсетевой протокол контрольных сообщений, который используется для передачи управляющих сообщений и информации об ошибках между системами. Например, сообщение «хост недоступен» передается по протоколу ICMP, равно как запросы и ответы, формируемые утилитой* `ping`. *IGMP (Internet Group Management Protocol) – это межсетевой протокол управления группами, с помощью которого хосты сообщают маршрутизаторам, поддерживающим групповое вещание, о принадлежности к локальным группам. Хотя сообщения протоколов ICMP и IGMP передаются в виде IP-датаграмм, они рассматриваются как неотъемлемая часть IP, а не как протоколы более высокого уровня.*

Интерфейсный уровень отвечает за взаимодействие между компьютером и физическим сетевым оборудованием. Он приблизительно соответствует канальному и физическому уровням модели OSI. Интерфейсный уровень по-настоящему не описан в документации по архитектуре TCP/IP. Там сказано только, что он обеспечивает доступ к сетевой аппаратуре системно-зависимым способом.

Прежде чем закончить тему уровней в стеке TCP/IP, рассмотрим, как происходит общение между уровнями стека протоколов в компьютерах на разных концах сквозного соединения. На рис. 2.25 изображены два стека TCP/IP на компьютерах, между которыми расположено несколько маршрутизаторов.

Вы знаете, что приложение передает данные стеку протоколов. Потом они опускаются вниз по стеку, передаются по сети, затем поднимаются вверх по стеку протоколов компьютера на другом конце и наконец попадают в приложение. Но при этом каждый уровень стека работает так, будто на другом конце находится только этот уровень и ничего больше. Например, если в качестве приложения выступает FTP, то FTP-клиент «говорит» непосредственно с FTP-сервером, не имея сведений о том, что между ними есть TCP, IP и физическая сеть.

Это верно и для других уровней. Например, если на транспортном уровне используется протокол TCP, то он общается только с протоколом TCP на другом конце, не зная, какие еще протоколы и сети используются для поддержания «беседы». В идеале должно быть так: если уровень N посылает сообщение, то уровень N на другом конце принимает только его, а все манипуляции, произведенные над этим сообщением нижележащими уровнями, оказываются невидимыми.

Последнее замечание требует объяснения. На рис. 2.25 вы увидите, что транспортный уровень – самый нижний из сквозных уровней, то есть таких, связь между которыми устанавливается без посредников. Напротив, в «разговоре» на межсетевом уровне участвуют маршрутизаторы или полнофункциональные компьютеры, расположенные на маршруте сообщения.

Примечание *Предполагается наличие промежуточных маршрутизаторов, то есть сообщение не попадает сразу в конечный пункт.*

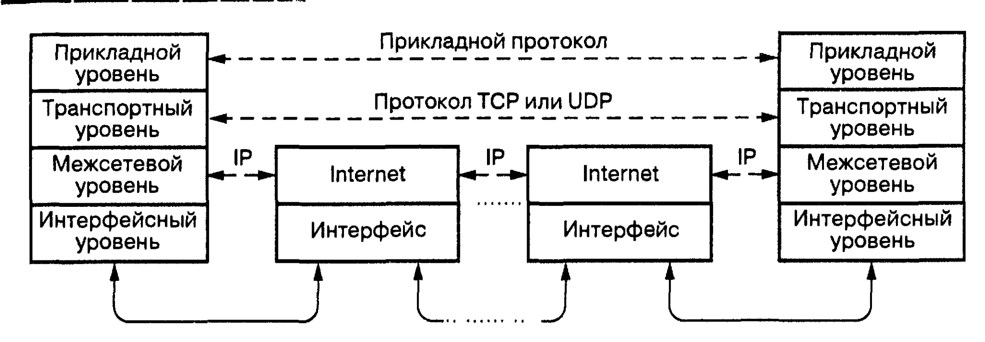


Рис. 2.25. Сквозная связь

Но промежуточные системы могут изменять некоторые поля, например, *время существования* датаграммы (TTL – time to live) в IP-заголовке. Поэтому межсетевой уровень в пункте назначения может «видеть» не в точности то же сообщение, что межсетевой уровень, который его послал.

Этот подчеркивает различие между межсетевым и транспортным уровнями. Межсетевой уровень отвечает за доставку сообщений в следующий узел на маршруте. И он общается с межсетевым уровнем именно этого узла, а не с межсетевым уровнем в конечной точке. Транспортные же уровни контактируют напрямую, не имея информации о существовании промежуточных систем.

Резюме

В этом разделе дано сравнение моделей OSI и TCP/IP. Вы узнали, что семиуровневая модель OSI нужна как средство описания сетевой архитектуры, но созданные на ее базе реализации почти не имеют успеха.

Глава 3. Создание эффективных и устойчивых сетевых программ

Совет 15. Разберитесь с операцией записи в TCP

Здесь и далее обсуждаются некоторые особенности операций чтения и записи при программировании TCP/IP. Представляет интерес не конкретный API и детали системных вызовов, а семантические вопросы, связанные с этими операциями.

Как сказано в совете 6, между операциями записи и посылаемыми TCP сегментами нет взаимно-однозначного соответствия. Как именно соотносятся обращения к операции записи с протоколом TCP, зависит от системы, но все же спецификации протокола достаточно определенны и можно сделать некоторые выводы при знакомстве с конкретной реализацией. Традиционная реализация подробно описана в системе BSD. Она часто рассматривается как эталонная, и ее исходные тексты доступны.

Примечание *Исходные тексты оригинальной реализации для системы 4.4BSD-lite2 можно получить на CD-ROM у компании Walnut Creek (http:// www.cdrom.com). Подробные пояснения к исходному тексту вы найдете в книге [Wright and Stevens 1995].*

Операция записи с точки зрения приложения

Когда пользователь выполняет запись в TCP-соединение, данные сначала копируются из буфера пользователя в память ядра. Дальнейшее зависит от состояния соединения. TCP может «решить», что надо послать все данные, только часть или ничего не посылать. О том, как принимается решение, будет сказано ниже. Сначала рассмотрим операцию записи с точки зрения приложения.

Хочется думать, что если операция записи n байт вернула значение n, то все эти n байт, действительно, переданы на другой конец и, возможно, уже подтверждены. Увы, это не так. TCP посылает столько данных, сколько возможно (или ничего), и немедленно возвращает значение n. Приложение не определяет, какая часть данных послана и были ли они подтверждены.

В общем случае операция записи не блокирует процесс, если только буфер передачи TCP не полон. Это означает, что после записи управление почти всегда быстро возвращается программе. После получения управления нельзя ничего гарантировать относительно местонахождения «записанных» данных. Как упоминалось в совете 9, это имеет значение для надежности передачи данных.

С точки зрения приложения данные записаны. Поэтому, помня о гарантиях доставки, предлагаемых TCP, можно считать, что информация дошла до другого конца. В действительности, некоторые (или все) эти данные в момент возврата из операции записи могут все еще стоять в очереди на передачу. И если хост или приложение на другом конце постигнет крах, то информация будет потеряна.

Примечание *Если отправляющее приложение завершает сеанс аварийно, то TCP все равно будет пытаться доставить данные.*

Еще один важный момент, который нужно иметь в виду, – это обработка ошибки записи. Если при записи на диск вызов `write` не вернул код ошибки, то точно известно, что запись была успешной.

Примечание *Строго говоря, это неверно. Обычно данные находятся в буфере в пространстве ядра до того момента, пока не произойдет сброс буферов на диск. Поэтому если до этого момента система «упадет», то данные вполне могут быть потеряны. Но суть в том что после возврата из `write` уже не будет никаких сообщений об ошибках. Можно признать потерю не сброшенных на диск данных неизбежной, но не более вероятной, чем отказ самого диска.*

При работе с TCP получение кода ошибки от операции записи – очень редкое явление. Поскольку операция записи возвращает управление до фактической отправки данных, обычно ошибки выявляются при последующих операциях, о чем говорилось в совете 9. Так как следующей операцией чаще всего бывает чтение предполагается, что ошибки записи обнаруживаются при чтении. Операция записи возвращает лишь ошибки, очевидные в момент вызова, а именно:

- неверный дескриптор сокета;
- файловый дескриптор указывает не на сокет (в случае вызова `send` и родственных функций);
- указанный при вызове сокет не существует или не подсоединен;
- в качестве адреса буфера указан недопустимый адрес.

Причина большинства этих проблем – ошибка в программе. После завершения стадии разработки они почти не встречаются. Исключение составляет код ошибки `EPIPE` (или сигнал `SIGPIPE`), который свидетельствует о сбросе соединения хостом на другом конце. Условия, при которых такая ошибка возникает, обсуждались в совете 9 при рассмотрении краха приложения-партнера.

Подводя итог этим рассуждениям, можно сказать, что применительно к TCP-соединениям операцию записи лучше представлять себе как копирование в очередь для передачи, сопровождаемое извещением TCP о появлении новых данных. Понятно, какую работу TCP произведет дальше, но эти действия будут асинхронны по отношению к самой записи.

Операция записи с точки зрения TCP

Как отмечалось выше, операция записи отвечает лишь за копирование данных из буфера приложения в память ядра и уведомление TCP о том, что появились

новые данные для передачи. А теперь рассмотрим некоторые из критериев, которыми руководствуется TCP, «принимая решение» о том, можно ли передать новые данные незамедлительно и в каком количестве. Я не задаюсь целью полностью объяснить логику отправки данных в TCP, а хочу лишь помочь вам составить представление о факторах, влияющих на эту логику. Тогда вы сможете лучше понять принципы работы своих программ.

Одна из основных целей стратегии отправки данных в TCP – максимально эффективное использование имеющейся полосы пропускания. TCP посылает данные блоками, размер которых равен MSS (maximum segment size – максимальный размер сегмента).

Примечание *В процессе установления соединения TCP на каждом конце может указать приемлемый для него MSS. TCP на другом конце обязан удовлетворить это пожелание и не посылать сегменты большего размера. MSS вычисляется на основе MTU (maxitimum transmission unit – максимальный размер передаваемого блока), как описано в совете 7.*

В то же время TCP не может переполнять буферы на принимающем конце. Как вы видели в совете 1, это определяется окном передачи.

Если бы эти два условия были единственными, то стратегия отправки была бы проста: немедленно послать все имеющиеся данные, упаковав их в сегменты размером MSS, но не более чем разрешено окном передачи. К сожалению, есть и другие факторы.

Прежде всего, очень важно не допускать перегрузки сети. Если TCP неожиданно пошлет в сеть большое число сегментов, может исчерпаться память маршрутизатора, что повлечет за собой отбрасывание датаграмм. А из-за этого начнутся повторные передачи, что еще больше загрузит сеть. В худшем случае сеть будет загружена настолько, что датаграммы вообще нельзя будет доставить. Это называется *затором* (congestion collapse). Чтобы избежать перегрузки, TCP не посылает по простаивающему соединению все сегменты сразу. Сначала он посылает один сегмент и постепенно увеличивает число неподтвержденных сегментов в сети, пока не будет достигнуто равновесие.

Примечание *Эту проблему можно наглядно проиллюстрировать таким примером. Предположим, что в комнате, полной народу, кто-то закричал: «Пожар!» Все одновременно бросаются к дверям, возникает давка, и в результате никто не может выйти. Если же люди будут выходить по одному, то пробки не возникнет, и все благополучно покинут помещение.*

Для предотвращения перегрузки TCP применяет два алгоритма, в которых используется еще одно окно, называемое *окном перегрузки*. Максимальное число байтов, которое TCP может послать в любой момент, – это минимальная из двух величин: размер окна передачи и размер окна перегрузки. Обратите внимание, что эти окна отвечают за разные аспекты управления потоком. Окно передачи, декларируемое

TCP на другом конце, предохраняет от переполнения его буферов. Окно перегрузки, отслеживаемое TCP на вашем конце, не дает превысить пропускную способность сети. Ограничив объем передачи минимальным из этих двух окон, вы удовлетворяете обоим требованиям управления потоком.

Первый алгоритм управления перегрузкой называется «медленный старт». Он постепенно увеличивает частоту передачи сегментов в сеть до пороговой величины.

Примечание *Слово «медленный» взято в кавычки, поскольку на самом деле нарастание частоты экспоненциально. При медленном старте окно перегрузки открывается на один сегмент при получении каждого ACK. Если вы начали с одного сегмента, то последовательные размеры окна будут составлять 1, 2, 4, 8 и т.д.*

Когда размер окна перегрузки достигает порога, который называется *порогом медленного старта*, этот алгоритм прекращает работу, и в дело вступает алгоритм *избежания перегрузки.* Его работа предполагает, что соединение достигло равновесного состояния, и сеть постоянно зондируется – не увеличилась ли пропускная способность. На этой стадии окно перегрузки открывается линейно – по одному сегменту за период кругового обращения.

В стратегии отправки TCP окно перегрузки в принципе может запретить посылать данные, которые в его отсутствие можно было бы послать. Если происходит перегрузка (о чем свидетельствует потерянный сегмент) или сеть некоторое время простаивает, то окно перегрузки сужается, возможно, даже до размера одного сегмента. В зависимости от того, сколько данных находится в очереди, и сколько их пытается послать приложение, это может препятствовать отправке всех данных.

Авторитетным источником информации об алгоритмах избежания перегрузки является работа [Jacobson 1988], в которой они впервые были предложены. Джекобсон привел результаты нескольких экспериментов, демонстрирующие заметное повышение производительности сети после внедрения управления перегрузкой. В книге [Stevens 1994] содержится подробное объяснение этих алгоритмов и результаты трассировки в локальной сети. В настоящее время эти алгоритмы следует включать в любую реализацию, согласующуюся со стандартом (RFC 1122 [Braden 1989]).

Примечание *Несмотря на впечатляющие результаты, реализация этих алгоритмов очень проста – всего две переменные состояния и несколько строчек кода. Детали можно найти в книге [Wright and Stevens 1995].*

Еще один фактор, влияющий на стратегию отправки TCP, – алгоритм Нейгла. Этот алгоритм впервые предложен в RFC 896 [Nagle 1984]. Он требует, чтобы никогда не было более одного неподтвержденного маленького сегмента, то есть сегмента размером менее MSS. Цель алгоритма Нейгла – не дать TCP забить сеть последовательностью мелких сегментов. Вместо этого TCP сохраняет в своих буферах небольшие блоки данных, пока не получит подтверждение на предыдущий маленький сегмент, после чего посылает сразу все накопившиеся данные. В совете 24 вы

увидите, что отключение алгоритма Нейгла может заметно сказаться на производительности приложения.

Если приложение записывает данные небольшими порциями, то эффект от алгоритма Нейгла очевиден. Предположим, что есть простаивающее соединение, окна передачи и перегрузки достаточно велики, а выполняются подряд две небольшие операции записи. Данные, записанные вначале, передаются немедленно, поскольку окна это позволяют, а алгоритм Нейгла не препятствует, так как неподтвержденных данных нет (соединение простаивало). Но, когда до TCP доходят данные, полученные при второй операции, они не передаются, хотя в окнах передачи и перегрузки есть место. Поскольку уже есть один неподтвержденный маленький сегмент, и алгоритм Нейгла требует оставить данные в очереди, пока не придет ACK.

Обычно при реализации алгоритма Нейгла не посылают маленький сегмент, если есть неподтвержденные данные. Такая процедура рекомендована RFC 1122. Но реализация в BSD (и некоторые другие) несколько отходит от этого правила и отправляет маленький сегмент, если это последний фрагмент большой одновременно записанной части данных, а соединение простаивает. Например, MSS для простаивающего соединения равен 1460 байт, а приложение записывает 1600 байт. При этом TCP пошлет (при условии, что это разрешено окнами передачи и перегрузки) сначала сегмент размером 1460, а сразу вслед за ним, не дожидаясь подтверждения, сегмент размером 140. При строгой интерпретации алгоритма Нейгла следовало бы отложить отправку второго сегмента либо до подтверждения первого, либо до того, как приложение запишет достаточно данных для формирования полного сегмента.

Алгоритм Нейгла – это лишь один из двух алгоритмов, позволяющих избежать *синдрома безумного окна* (SWS – silly window syndrome). Смысл этой тактики в том, чтобы не допустить отправки небольших объемов данных. Синдром SWS и его отрицательное влияние на производительность обсуждаются в RFC 813 [Clark 1982]. Как вы видели, алгоритм Нейгла пытается избежать синдрома SWS со стороны отправителя. Но требуются и усилия со стороны получателя, который не должен декларировать слишком маленькие окна.

Напомним, что окно передачи дает оценку свободного места в буферах хоста на другом конце соединения. Этот хост объявляет о том, сколько в нем имеется места, включая в каждый посылаемый сегмент информацию об *обновлении окна*. Чтобы избежать SWS, получатель не должен объявлять о небольших изменениях.

Следует пояснить это на примере. Предположим, у получателя есть 14600 свободных байт, а MSS составляет 1460 байт. Допустим также, что приложением на конце получателя читается за один раз всего по 100 байт. Отправив получателю 10 сегментов, окно передачи закроется. И вы будете вынуждены приостановить отправку данных. Но вот приложение прочитало 100 байт, в буфере приема 100 байт освободилось. Если бы получатель объявил об этих 100 байтах, то вы тут же послали бы ему маленький сегмент, поскольку TCP временно отменяет алгоритм Нейгла, если из-за него длительное время невозможно отправить маленький сегмент. Вы и дальше продолжали бы посылать стобайтные пакеты, так как всякий раз, когда приложение на конце получателя читает очередные 100 байт, получатель объявляет об освобождении этих 100 байт, посылая информацию об обновлении окна.

Алгоритм избежания синдрома SWS на получающем конце не позволяет объявлять об обновлении окна, если объем буферной памяти значительно не увеличился. В RFC 1122 «значительно» – это на размер полного сегмента или более чем на половину максимального размера окна. В реализациях, производных от BSD, требуется увеличение на два полных сегмента или на половину максимального размера окна.

Может показаться, что избежание SWS со стороны получателя излишне (поскольку отправителю не разрешено посылать маленькие сегменты), но в действительности это защита от тех стеков TCP/IP, в которых алгоритм Нейгла не реализован или отключен приложением (совет 24). RFC 1122 требует от реализаций TCP, удовлетворяющих стандарту, осуществлять избежание SWS на обоих концах.

На основе этой информации теперь можно сформулировать стратегию отправки, принятую в реализациях TCP, производных от BSD. В других реализациях стратегия может быть несколько иной, но основные принципы сохраняются.

При каждом вызове процедуры вывода TCP вычисляет объем данных, которые можно послать. Это минимальное значение количества данных в буфере передачи, размера окон передачи и перегрузки и MSS. Данные отправляются при выполнении хотя бы одного из следующих условий:

- можно послать полный сегмент размером MSS;
- соединение простаивает, и можно опустошить буфер передачи;
- алгоритм Нейгла отключен, и можно опустошить буфер передачи;
- есть срочные данные для отправки;
- есть маленький сегмент, но его отправка уже задержана на достаточно длительное время;

Примечание *Если у TCP есть маленький сегмент, который запрещено посылать, то он взводит таймер на то время, которое потребовалось бы для ожидания ACK перед повторной передачей (но в пределах 5–60 с). Иными словами, устанавливается тайм-аут ретрансмиссии (RTO). Если этот таймер, называемый таймером терпения (persist timer), срабатывает, то TCP все-таки посылает сегмент при условии, что это не противоречит ограничениям, которые накладывают окна передачи и перегрузки. Даже если получатель объявляет окно размером нуль байт, TCP все равно попытается послать один байт. Это делается для того, чтобы потерянное обновление окна не привело к тупиковой ситуации.*

- окно приема, объявленное хостом на другом конце, открыто не менее чем наполовину;
- необходимо повторно передать сегмент;
- требуется послать ACK на принятые данные;
- нужно объявить об обновлении окна.

Резюме

В этом разделе подробно рассмотрена операция записи. С точки зрения приложения операцию записи проще всего представлять как копирование из адресного

пространства пользователя в буферы ядра и последующий возврат. Срок передачи данных TCP и их объем зависят от состояния соединения, а приложение не имеет подобной информации.

Проанализированы стратегия отправки, принятая в BSD TCP, а также влияние на нее объема буферной памяти у получателя (представлен окном передачи), оценки загрузки сети (представлена окном перегрузки), объема данных, готовых для передачи, попытки избежать синдрома безумного окна и стратегии повторной передачи.

Совет 16. Разберитесь с аккуратным размыканием TCP-соединений

Как вы уже видели, в работе TCP-соединения есть три фазы:

1. Установления соединения.
2. Передачи данных.
3. Разрыва соединения.

В этом разделе будет рассмотрен переход от фазы передачи данных к фазе разрыва соединения. Точнее, как узнать, что хост на другом конце завершил фазу передачи данных и готов к разрыву соединения, и как он может сообщить об этом партнеру.

Вы увидите, что один хост может прекратить отправку данных и сигнализировать партнеру об этом, не отказываясь, однако, от приема данных. Это возможно, поскольку TCP-соединения полнодуплексные, потоки данных в разных направлениях не зависят друг от друга.

Например, клиент может соединиться с сервером, отправить серию запросов, а затем закрыть свою половину соединения, предоставив тем самым серверу информацию, что больше запросов не будет. Серверу для ответа клиенту, возможно, понадобится выполнить большой объем работы и даже связаться с другими серверами, так что он продолжает посылать данные уже после того, как клиент прекратил отправлять запросы. С другой стороны, сервер может послать в ответ сколько угодно данных, так что клиент не определяет заранее, когда ответ закончится. Поэтому сервер, вероятно, как и клиент, закроет свой конец соединения, сигнализируя о конце передачи.

После того как ответ на последний запрос клиента отправлен и сервер закрыл свой конец соединения, TCP завершает фазу разрыва. Обратите внимание, что закрытие соединения рассматривается как естественный способ известить партнера о прекращении передачи данных. По сути, посылается признак конца файла EOF.

Вызов shutdown

Как приложение закрывает свой конец соединения? Оно не может просто завершить сеанс или закрыть сокет, поскольку у партнера могут быть еще данные. В API сокетов есть интерфейс `shutdown`. Он используется так же, как и вызов `close`, но при этом передается дополнительный параметр, означающий, какую сторону соединения надо закрыть.

```
#include <sys/socket.h>  /* UNIX. */
#include <winsock2.h>    /* Windows. */

int shutdown( int s, int how );         /* UNIX. */
int shutdown( SOCKET s, int how );      /* Windows. */
```

Возвращаемое значение: 0 – нормально, –1 (UNIX) или SOCKET_ERROR (Windows) – ошибка.

К сожалению, между реализациями shutdown в UNIX и Windows есть различия в семантике и API. Традиционно в качестве значений параметра *how* вызова shutdown использовались числа. И в стандарте POSIX, и в спецификации Winsock им присвоены символические имена, только разные. В табл. 3.1 приведены значения, символические константы для них и семантика параметра *how*.

Различия в символических именах можно легко компенсировать, определив в заголовочном файле одни константы через другие или используя числовые значения. А вот семантические отличия гораздо серьезнее. Посмотрим, для чего предназначено каждое значение.

Таблица 3.1. Значения параметра how для вызова shutdown

	Значение how		
Числовое	**POSIX**	**Winsock**	**Действие**
0	SHUT_RD	SD_RECEIVE	Закрывается принимающая сторона соединения
1	SHUT_WR	SD_SEND	Закрывается передающая сторона соединения
2	SHUT_RDWR	SD_BOTH	Закрываются обе стороны

how = 0 Закрывается принимающая сторона соединения. В обеих реализациях в сокете делается пометка, что он больше не может принимать данные и должен вернуть EOF, если приложением делаются попытки еще что-то читать. Но отношение к данным, уже находившимся в очереди приложения в момент выполнения shutdown, а также к приему новых данных от хоста на другом конце различное. В UNIX все ранее принятые, но еще не прочитанные данные уничтожаются, так что приложение их уже не получит. Если поступают новые данные, то TCP их подтверждает и тут же отбрасывает, поскольку приложение не хочет принимать новые данные. Наоборот, в соответствии с Winsock соединение вообще разрывается, если в очереди есть еще данные или поступают новые. Поэтому некоторые авторы (например, [Quinn and Shute 1996]) считают, что под Windows использование конструкции

```
shutdown( s, 0 );
```

небезопасно.

how = 1 Закрывается отправляющая сторона соединения. В сокете делается пометка, что данные посылаться больше не будут; все последующие попытки выполнить для него операцию записи заканчиваются ошибкой. После того как вся информация из буфера отправлена, TCP посылает

сегмент FIN, сообщая партнеру, что данных больше не будет. Это называется *полузакрытием* (half close). Такое использование вызова shutdown наиболее типично, и его семантика в обеих реализациях одинакова.

how = 2 Закрываются обе стороны соединения. Эффект такой же, как при выполнении вызова shutdown дважды, один раз с *how* = 0, а другой – с *how* = 1. Хотя, на первый взгляд, обращение

```
shutdown( s, 2 );
```

эквивалентно вызову close или closesocket, в действительности это не так. Обычно нет причин для вызова shutdown с параметром *how* = 2, но в работе [Quinn and Shute 1996] сообщается, что в некоторых реализациях Winsock вызов closesocket работает неправильно, если предварительно не было обращения к shutdown с *how* = 2. В соответствии с Winsock вызов shutdown с *how* = 2 создает ту же проблему, что и вызов с *how* = 0, – может быть разорвано соединение.

Между закрытием сокета и вызовом shutdown есть существенные различия. Во-первых, shutdown не закрывает сокет по-настоящему, даже если он вызван с параметром 2. Иными словами, ни сокет, ни ассоциированные с ним ресурсы (за исключением буфера приема, если *how* = 0 или 2) не освобождаются. Кроме того, воздействие shutdown распространяется на все процессы, в которых этот сокет открыт. Так, например, вызов shutdown с параметром *how* = 1 делает невозможной запись в этот сокет для всех его владельцев. При вызове же close или closesocket все остальные процессы могут продолжать пользоваться сокетом.

Последний факт во многих случаях можно обратить на пользу. Вызывая shutdown с *how* = 1, будьте уверены, что партнер получит EOF, даже если этот сокет открыт и другими процессами. При вызове close или closesocket это не гарантируется, поскольку TCP не пошлет FIN, пока счетчик ссылок на сокет не станет равным нулю. А это произойдет только тогда, когда все процессы закроют этот сокет.

Наконец, стоит упомянуть, что, хотя в этом разделе говорится о TCP, вызов shutdown применим и к UDP. Поскольку нет соединения, которое можно закрыть, польза обращения к shutdown с *how* = 1 или 2, остается под вопросом, но задавать параметр *how* = 0 можно для предотвращения приема датаграмм из конкретного UDP-порта.

Аккуратное размыкание соединений

Теперь, когда вы познакомились с вызовом shutdown, посмотрите, как его можно использовать для *аккуратного размыкания* соединения. Цель этой операции – гарантировать, что обе стороны получат все предназначенные им данные до того, как соединение будет разорвано.

Примечание *Термин «аккуратное размыкание» (orderly release) имеет некоторое отношение к команде* t_sndrel *из API XTI (совет 5), которую также часто называют аккуратным размыканием в отличие от команды грубого размыкания (abortive release)* t_snddis. *Но*

путать их не стоит. Команда `t_sndrel` выполняет те же действия, что и `shutdown`. Обе команды используются для аккуратного размыкания соединения.

Просто закрыть соединение в некоторых случаях недостаточно, поскольку могут быть потеряны еще не принятые данные. Помните, что, когда приложение закрывает соединение, недоставленные данные отбрасываются.

Чтобы поэкспериментировать с аккуратным размыканием, запрограммируйте клиент, который посылает серверу данные, а затем читает и печатает ответ сервера. Текст программы приведен в листинге 3.1. Клиент читает из стандартного входа данные для отправки серверу. Как только `fgets` вернет `NULL`, индицирующий конец файла, клиент начинает процедуру разрыва соединения. Параметр `-c` в командной строке управляет этим процессом. Если `-c` не задан, то программа `shutdownc` вызывает `shutdown` для закрытия передающего конца соединения. Если же параметр задан, то `shutdownc` вызывает `CLOSE`, затем пять секунд «спит» и завершает сеанс.

Листинг 3.1. Клиент для экспериментов с аккуратным размыканием

shutdownc.c

```
#include "etcp.h"
int main( int argc, char **argv )
{
  SOCKET s;
  fd_set readmask;
  fd_set allreads;
  int rc;
  int len;
  int c;
  int closeit = FALSE;
  int err = FALSE;
  char lin[ 1024 ];
  char lout[ 1024 ];

  INIT();
  opterr = FALSE;
  while ( ( c = getopt( argc, argv, "c" ) ) != EOF )
  {
     switch( c )
     {
        case 'c' :
           closeit = TRUE;
           break;
        case '?' :
           err = TRUE;
     }
  }
  if ( err || argc - optind != 2 )
     error( 1, 0, "Порядок вызова: %s [-c] хост порт\n",
        program_name );
```

```
30    s = tcp_client( argv[ optind ], argv[ optind + 1 ] );
31    FD_ZERO( &allreads );
32    FD_SET( 0, &allreads );
33    FD_SET( s, &allreads );
34    for ( ;; )
35    {
36       readmask = allreads;
37       rc = select( s + 1, &readmask, NULL, NULL, NULL );
38       if ( rc <= 0 )
39          error( 1, errno, "ошибка: select вернул (%d)", rc );
40       if ( FD_ISSET( s, &readmask ) )
41       {
42          rc = recv( s, lin, sizeof( lin ) - 1, 0 );
43          if ( rc < 0 )
44             error( 1, errno, "ошибка вызова recv" );
45          if ( rc == 0 )
46             error( 1, 0, "сервер отсоединился\n" );
47          lin[ rc ] = '\0';
48          if ( fputs( lin, stdout ) == EOF )
49             error( 1, errno, "ошибка вызова fputs" );
50       }
51       if ( FD_ISSET( 0, &readmask ) )
52       {
53          if ( fgets( lout, sizeof( lout ), stdin ) == NULL )
54          {
55             FD_CLR( 0, &allreads );
56             if ( closeit )
57             {
58                CLOSE( s );
59                sleep( 5 );
60                EXIT( 0 );
61             }
62             else if ( shutdown( s, 1 ) )
63                error( 1, errno, "ошибка вызова shutdown" );
64          }
65          else
66          {
67             len = strlen( lout );
68             rc = send( s, lout, len, 0 );
69             if ( rc < 0 )
70                error( 1, errno, "ошибка вызова send" );
71          }
72       }
73    }
74 }
```

shutdownc.c

Инициализация

14-30 Выполняем обычную инициализацию клиента и проверяем, есть ли в командной строке флаг -c.

Обработка данных

40-50 Если в TCP-сокете есть данные для чтения, программа пытается прочитать, сколько можно, но не более, чем помещается в буфер. При получении признака конца файла или ошибки завершаем сеанс, в противном случае выводим все прочитанное на `stdout`.

Примечание *Обратите внимание на конструкцию `sizeof( lin ) - 1` в вызове `recv` на строке 42. Вопреки всем призывам избегать переполнения буфера, высказанным в совете 11, в первоначальной версии этой программы было написано `sizeof( lin )`, что привело к записи за границей буфера в операторе*

```
lin[ rc ] = '\0';
```

в строке 47.

53-64 Прочитав из стандартного входа EOF, вызываем либо `shutdown`, либо `CLOSE` в зависимости от наличия флага `-c`.

65-71 В противном случае передаем прочитанные данные серверу.

Можно было бы вместе с этим клиентом использовать стандартный системный сервис эхо-контроля, но, чтобы увидеть возможные ошибки и ввести некоторую задержку, напишите собственную версию эхо-сервера. Ничего особенного в программе `tcpecho.c` нет. Она только распознает дополнительный аргумент в командной строке, при наличии которого программа «спит» указанное число секунд между чтением и записью каждого блока данных (листинг 3.2).

Сначала запустим клиент `shutdownc` с флагом `-c`, чтобы он закрывал сокет после считывания EOF из стандартного ввода. Поставим в сервере `tcpecho` задержку на 4 с перед отправкой назад только прочитанных данных:

```
bsd: $ tcpecho 9000 4 &
[1] 3836
bsd: $ shutdownc -c localhost 9000
data1                                       Эти три строки были введены
data2                                       подряд максимально быстро
^D
tcpecho: ошибка вызова send: Broken pipe (32)     Спустя 4 с
                                                  после отправки "data1".
```

Листинг 3.2. Эхо-сервер на базе TCP

```
                                                              tcpecho.c
#include "etcp.h"

int main( int argc, char **argv )
{
    SOCKET s;
    SOCKET s1;
    char buf[ 1024 ];
    int rc;
```

```
    int nap = 0;

    INIT();
    if ( argc == 3 )
        nap = atoi( argv[ 2 ] );
    s = tcp_server( NULL, argv[ 1 ] );
    s1 = accept( s, NULL, NULL );
    if ( !isvalidsock( s1 ) )
        error( 1, errno, "ошибка вызова accept" );
    signal( SIGPIPE, SIG_IGN ); /* Игнорировать сигнал SIGPIPE. */
    for ( ;; )
    {
        rc = recv( s1, buf, sizeof( buf ), 0 );
        if ( rc == 0 )
            error( 1, 0, "клиент отсоединился\n" );
        if ( rc < 0 )
            error( 1, errno, "ошибка вызова recv" );
        if ( nap )
            sleep( nap );
        rc = send( s1, buf, rc, 0 );
        if ( rc < 0 )
            error( 1, errno, "ошибка вызова send" );
    }
}
```
tcpecho.c

Затем нужно напечатать две строки **data1** и **data2** и сразу вслед за ними нажать комбинацию клавиш **Ctrl+D**, чтобы послать программе shutdownc конец файла и вынудить ее закрыть сокет. Заметьте, что сервер не вернул ни одной строки. В напечатанном сообщении tcpecho об ошибке говорится, что произошло. Когда сервер вернулся из вызова sleep и попытался отослать назад строку data1, он получил RST, поскольку клиент уже закрыл соединение.

Примечание *Как объяснялось в совете 9, ошибка возвращается при записи второй строки (data2). Заметьте, что это один из немногих случаев, когда ошибку возвращает операция записи, а не чтения. Подробнее об этом рассказано в совете 15.*

В чем суть проблемы? Хотя клиент сообщил серверу о том, что больше не будет посылать данные, но соединение разорвал до того, как сервер успел завершить обработку, в результате информация была потеряна. В левой половине рис. 3.2 показано, как происходил обмен сегментами.

Теперь повторим эксперимент, но на этот раз запустим shutdownc без флага -c.

```
bsd: $ tcpecho 9000 4 &
[1] 3845
bsd: $ shutdownc localhost 9000
data1
data2
^D
```

```
data1                      Спустя 4 с после отправки "data1".
data2                      Спустя 4 с после получения "data1".
tcpecho: клиент отсоединился
shutdownc: сервер отсоединился
```

На этот раз все сработало правильно. Прочитав из стандартного входа признак конца файла, shutdownc вызывает shutdown, сообщая серверу, что он больше не будет ничего посылать, но продолжает читать данные из соединения. Когда сервер tcpecho обнаруживает EOF, посланный клиентом, он закрывает соединение, в результате чего TCP посылает все оставшиеся в очереди данные, а вместе с ними FIN. Клиент, получив EOF, определяет, что сервер отправил все, что у него было, и завершает сеанс.

Заметьте, что у сервера нет информации, какую операцию (shutdown или close) выполнит клиент, пока не попытается писать в сокет и не получит код ошибки или EOF. Как видно из рис. 3.1, оба конца обмениваются теми же сегментами, что и раньше, до того, как TCP клиента ответил на сегмент, содержащий строку **data1**.

Стоит отметить еще один момент. В примерах вы несколько раз видели, что, когда TCP получает от хоста на другом конце сегмент FIN, он сообщает об этом приложению, возвращая нуль из операции чтения. Примеры приводятся в строке 45 листинга 3.1 и в строке 20 листинга 3.2, где путем сравнения кода возврата recv с нулем проверяется, получен ли EOF. Часто возникает путаница, когда в ситуации подобной той, что показана в листинге 3.1, используется системный вызов select Когда приложение на другом конце закрывает отправляющую сторону соединения, вызывая close или shutdown либо просто завершая работу, select возвращает управление, сообщая, что в сокете есть данные для чтения. Если приложение при этом не проверяет EOF, то оно может попытаться обработать сегмент нулевой длины или зациклиться, переключаясь между вызовами read и select.

В сетевых конференциях часто отмечают, что «select свидетельствует о наличии информации для чтения, но в действительности ничего не оказывается». В действительности хост на другом конце просто закрыл, как минимум, отправляющую сторону соединения, и данные, о присутствии которых говорит select, – это всего лишь признак конца файла.

Резюме

Вы изучили системный вызов shutdown и сравнили его с вызовом close. Также рассказывалось, что с помощью shutdown можно закрыть только отправляющую, принимающую или обе стороны соединения; и счетчик ссылок на сокет при этом изменяется иначе, чем при закрытии с помощью close.

Затем было показано, как использовать shutdown для аккуратного размыкания соединения. Аккуратное размыкание – это последовательность разрыва соединения, при которой данные не теряются.

Совет 17. Подумайте о запуске своего приложения через inetd

В операционной системе UNIX и некоторых других имеется сетевой суперсервер inetd, который позволяет почти без усилий сделать приложение сетевым

shutdownc tcpecho

data 1
<ACK>
data 2
<ACK>
close
<FIN>
<ACK>
data 1
<RST>

Соединение закрыто

shutdownc tcpecho

data 1
<ACK>
data 2
<ACK>
shutdown
<FIN>
<ACK>
data 1
<ACK>
data 2
<ACK>
<FIN>
<ACK>

Соединение разомкнуто с помощью shutdown

Рис. 3.1. Системные вызовы close и shutdown

Кроме того, если есть всего один процесс, который прослушивает входящие соединения и входящие UDP-датаграммы, то можно сэкономить системные ресурсы.

Обычно `inetd` поддерживает, по меньшей мере, протоколы TCP и UDP, а возможно, и некоторые другие. Здесь будут рассмотрены только два первых. Поведение `inetd` существенно зависит от того, с каким протоколом – TCP или UDP – он работает.

TCP-серверы

Для TCP-серверов `inetd` прослушивает хорошо известные порты, ожидая запроса на соединение, затем принимает соединение, ассоциирует с ним файловые дескрипторы `stdin`, `stdout` и `stderr`, после чего запускает приложение. Таким образом, сервер может работать с соединением через дескрипторы 0, 1 и 2. Если это допускается конфигурационным файлом `inetd` (`/etc/inetd.conf`), то `inetd`

продолжает прослушивать тот же порт. Когда в этот порт поступает запрос на новое соединение, запускается новый экземпляр сервера, даже если первый еще не завершил сеанс. Это показано на рис. 3.2. Обратите внимание, что серверу не нужно обслуживать нескольких клиентов. Он просто выполняет запросы одного клиента, а потом завершается. Остальные клиенты обслуживаются дополнительными экземплярами сервера.

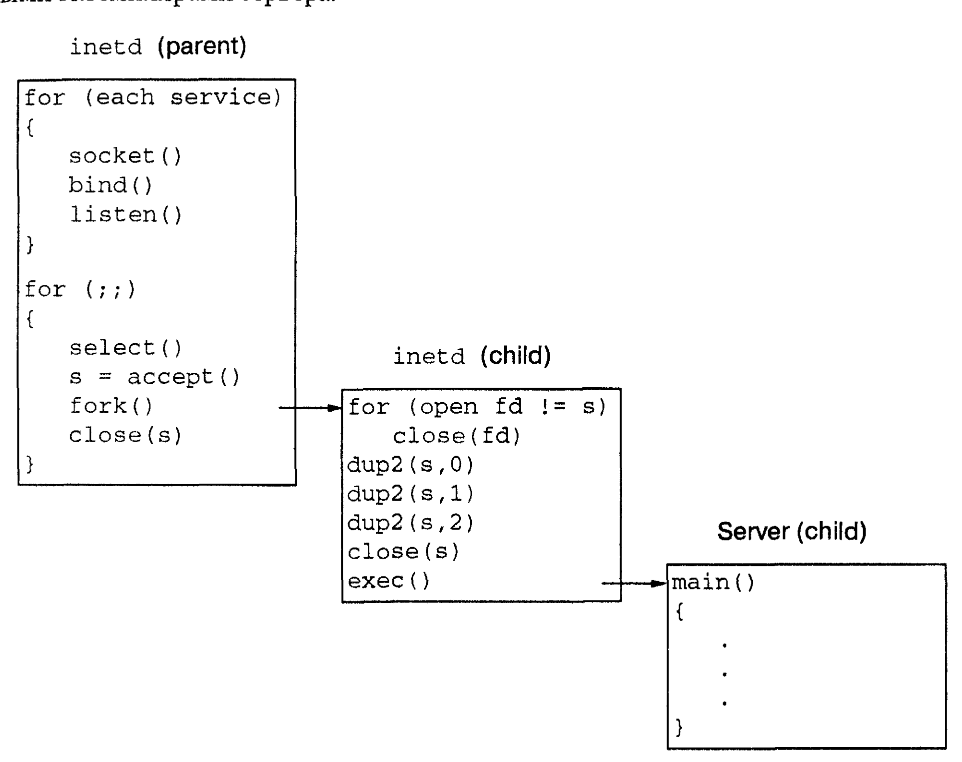


Рис. 3.2. Действия inetd при запуске TCP-сервера

Применение inetd освобождает от необходимости самостоятельно устанавливать TCP или UDP-соединение и позволяет писать сетевое приложение почти так же, как обычный фильтр. Простой, хотя и не очень интересный пример приведен в листинге 3.3.

Листинг 3.3. Программа rlnumd для подсчета строк

rlnumd.c

```
#include <stdio.h>
void main( void )
{
    int cnt = 0;
    char line[ 1024 ];
    /*
     * Мы должны явно установить режим построчной буферизации,
     * так как функции из библиотеки стандартного ввода/вывода
```

```
   * не считают сокет терминалом. */
  setvbuf( stdout, NULL, _IOLBF, 0 );
  while ( fgets( line, sizeof( line ), stdin ) != NULL )
     printf( "%3i: %s", ++cnt, line );
}
```
rlnumd.c

По поводу этой программы стоит сделать несколько замечаний:

- в тексте программы не упоминается ни о TCP, ни вообще о сети. Это не значит, что нельзя выполнять связанные с сокетами вызовы (`getpeername`, `[gs]etsockopt` и т.д.), просто в этом не всегда есть необходимость. Нет никаких ограничений и на использование `read` и `write`. Кроме того, можно пользоваться вызовами `send`, `recv`, `sendto` и `recvfrom`, как если бы `inetd` не было.
- режим буферизации строк приходится устанавливать самостоятельно, поскольку стандартная библиотека ввода/вывода автоматически устанавливает подобный режим только в том случае, если считает, что вывод производится на терминал. Это позволяет обеспечить быстрое время реакции для интерактивных приложений;
- стандартная библиотека берет на себя разбиение входного потока на строки. Об этом уже говорилось в совете 6;
- предполагаем, что не будет строк длиннее 1023 байт. Более длинные строки будут разбиты на несколько частей, и у каждой будет свой номер;

Примечание *Этот факт, который указан в книге [Oliver 2000], служит еще одним примером того, как можно легко допустить ошибку переполнения буфера. Подробнее этот вопрос обсуждался в совете 11.*

- хотя это приложение тривиально, но во многих «настоящих» TCP-приложениях, например telnet, rlogin и ftp, используется такая же техника.

Программа в листинге 3.3 может работать и как «нормальный» фильтр, и как удаленный сервис подсчета строк. Чтобы превратить ее в удаленный сервис, нужно только выбрать номер порта, добавить в файл `/etc/services` строку с именем сервиса и номером порта и включить в файл `/etc/inetd.conf` строку, описывающую этот сервис и путь к исполняемой программе. Например, если вы назовете сервис `rlnum`, исполняемую программу для него – `rlnumd` и назначите ему порт 8000, то надо будет добавить в `/etc/services` строку

```
rlnum 8000/tcp     # удаленный сервис подсчета строк,
```

а в `/etc/inetd.conf` – строку

```
rlnum stream tcp nowait jcs /usr/home/jcs/rlnumd rlnumd.
```

Добавленная в `/etc/services` строка означает, что сервис `rlnum` использует протокол TCP по порту 8000. Смысл же полей в строке, добавленной в `/etc/inetd.conf`, таков:

- имя сервиса, как он назван в `/etc/services`. Это имя хорошо известного порта, к которому подсоединяются клиенты данного сервера. В вашем примере – `rlnum`;

- тип сокета, который нужен серверу. Для TCP-серверов это stream, а для UDP-серверов – dgram. Поскольку здесь сервер пользуется протоколом TCP, указан stream;
- протокол, применяемый с сервером, – tcp или udp. В данном примере это tcp;
- флаг wait/nowait. Для UDP-серверов его значение *всегда* wait, а для TCP-серверов – почти всегда nowait. Если задан флаг nowait, то inetd сразу после запуска сервера возобновляет прослушивание связанного с ним хорошо известного порта. Если же задан флаг wait, то inetd не производит никакой работы с этим сокетом, пока сервер не завершится. А затем он возобновляет прослушивание порта в ожидании запросов на новые соединения (для stream-серверов) или новых датаграмм (для dgram-серверов). Если для stream-сервера задан флаг wait, то inetd не вызывает accept для соединения, а передает сокет, находящийся в режиме прослушивания, самому серверу, который должен принять хотя бы одно соединение перед завершением. Как отмечено в сообщении [Kacker 1998], задание флага wait для TCP-приложения – это мощная, но редко используемая возможность. Здесь приводится несколько применений флага wait для TCP-соединений:
  - в качестве механизма рестарта для ненадежных сетевых программ-демонов. Пока демон работает корректно, он принимает соединения от клиентов, но если по какой-то причине демон «падает», то при следующей попытке соединения inetd его рестартует;
  - как способ гарантировать одновременное подключение только одного клиента;
  - как способ управления многопоточным или многопроцессным приложением, зависящим от нагрузки. В этом случае начальный процесс запускается inetd, а затем он динамически балансирует нагрузку, создавая по мере необходимости дополнительные процессы или потоки. При уменьшении нагрузки, потоки уничтожаются, а в случае длительного простоя завершает работу и сам процесс, освобождая ресурсы и возвращая прослушивающий сокет inetd.

  В данном примере задан флаг nowait, как и обычно для TCP-серверов.
- имя пользователя, с правами которого будет запущен сервер. Это имя должно присутствовать в файле /etc/passwd. Большинство стандартных серверов, прописанных в inetd.conf, запускаются от имени root, но это совершенно необязательно. Здесь в качестве имени пользователя выбрано jcs.
- полный путь к файлу исполняемой программы. Поскольку rlnumd находится в каталоге пользователя jcs, задан путь /usr/home/jcs/rlnumd;
- до пяти аргументов (начиная с argv[0]), которые будут переданы серверу. Поскольку в этом примере у сервера нет аргументов, оставлен только argv[0].

Чтобы протестировать сервер, необходимо заставить inetd перечитать свой конфигурационный файл (в большинстве реализаций для этого нужно послать ему сигнал SIGHUP) и соединиться с помощью telnet:

```
bsd: $ telnet localhost rlnum
Trying 127.0.0.1...
Connected to localhost
```

```
Escape character is "^]".
hello
  1: hello
world
  2: world
^]
telnet> quit
Connection closed.
bsd: $
```

UDP-серверы

Поскольку в протоколе UDP соединения не устанавливаются (совет 1), `inetd` нечего слушать. При этом `inetd` запрашивает операционную систему (с помощью вызова `select`) о приходе новых датаграмм в порт UDP-сервера. Получив извещение, `inetd` дублирует дескриптор сокета на `stdin`, `stdout` и `stderr` и запускает UDP-сервер. В отличие от работы с TCP-серверами при наличии флага `nowait`, `inetd` больше не предпринимает с этим портом никаких действий, пока сервер не завершит сеанс. В этот момент он снова предлагает системе извещать его о новых датаграммах. Прежде чем закончить работу, серверу нужно прочесть хотя бы одну датаграмму из сокета, чтобы `inetd` не «увидел» то же самое сообщение, что и раньше. В противном случае он опять запустит сервер, войдя в бесконечный цикл.

Пример простого UDP-сервера, запускаемого через `inetd`, приведен в листинге 3.4. Этот сервер возвращает то, что получил, добавляя идентификатор своего процесса.

Листинг 3.4. Простой сервер, реализующий протокол запрос-ответ

udpecho1.c

```
 1 #include "etcp.h"
 2 int main( int argc, char **argv )
 3 {
 4    struct sockaddr_in peer;
 5    int rc;
 6    int len;
 7    int pidsz;
 8    char buf[ 120 ];

 9    pidsz = sprintf( buf, "%d: ", getpid() );
10    len = sizeof( peer );
11    rc = recvfrom( 0, buf + pidsz, sizeof( buf ) - pidsz, 0,
12       ( struct sockaddr * )&peer, &len );
13    if ( rc <= 0 )
14       exit( 1 );
15    sendto( 1, buf, rc + pidsz, 0,
16       ( struct sockaddr * )&peer, len );
17    exit( 0 );
18 }
```

udpecho1.c

updecho1

9 Получаем идентификатор процесса сервера (PID) от операционной системы, преобразуем его в код ASCII и помещаем в начало буфера ввода/вывода.

10-14 Читаем датаграмму от клиента и размещаем ее в буфере после идентификатора процесса.

15-17 Возвращаем клиенту ответ и завершаем сеанс.

Для экспериментов с этим сервером воспользуемся простым клиентом, код которого приведен в листинге 3.5. Он читает запросы из стандартного ввода, отсылает их серверу и печатает ответы на стандартном выводе.

Листинг 3.5. Простой UDP-клиент

udpclient.c

```
 1 #include "etcp.h"
 2 int main( int argc, char **argv )
 3 {
 4    struct sockaddr_in peer;
 5    SOCKET s;
 6    int rc = 0;
 7    int len;
 8    char buf[ 120 ];
 9    INIT();
10    s = udp_client( argv[ 1 ], argv[ 2 ], &peer );
11    while ( fgets( buf, sizeof( buf ), stdin ) != NULL )
12    {
13       rc = sendto( s, buf, strlen( buf ), 0,
14          ( struct sockaddr * )&peer, sizeof( peer ) );
15       if ( rc < 0 )
16          error( 1, errno, "ошибка вызова sendto" );
17       len = sizeof( peer );
18       rc = recvfrom( s, buf, sizeof( buf ) - 1, 0,
19          ( struct sockaddr * )&peer, &len );
20       if ( rc < 0 )
21          error( 1, errno, "ошибка вызова recvfrom" );
22       buf[ rc ] = '\0';
23       fputs( buf, stdout );
24    }
25    EXIT( 0 );
26 }
```

udpclient.c

10 Вызываем функцию `udp_client`, чтобы она поместила в структуру `peer` адрес сервера и получила UDP-сокет.

11-16 Читаем строку из стандартного ввода и посылаем ее в виде UDP-датаграммы хосту и в порт, указанные в командной строке.

17-21 Вызываем `recvfrom` для чтения ответа сервера и в случае ошибки завершаем сеанс.

22-23 Добавляем в конец ответа двоичный нуль и записываем строку на стандартный вывод.

В отношении программы udpclient можно сделать два замечания:

- в реализации клиента предполагается, что он всегда получит ответ от сервера. Как было сказано в совете 1, нет гарантии, что посланная сервером датаграмма будет доставлена. Поскольку udpclient – это интерактивная программа, ее всегда можно прервать и запустить заново, если она «зависнет» в вызове recvfrom. Но если бы клиент не был интерактивным, нужно было бы взвести таймер, чтобы предотвратить потери датаграмм;

Примечание *В сервере udpechol об этом не нужно беспокоиться, так как точно известно, что датаграмма уже пришла (иначе inetd не запустил бы сервер). Однако уже в следующем примере (листинг 3.6) приходится думать о потере датаграмм, так что таймер ассоциирован с recvfrom.*

- при работе с сервером udpechol не нужно получать адрес и порт отправителя, так как они уже известны. Поэтому строки 18 и 19 можно было бы заменить на:

```
rc = recvfrom( s, buf, sizeof( buf ) - 1, 0, NULL, NULL );
```

Но, как показано в следующем примере, иногда клиенту необходимо иметь информацию, с какого адреса сервер послал ответ, поэтому приведенные здесь UDP-клиенты всегда извлекают адрес.

Для тестирования сервера добавьте в файл /etc/inetd.conf на машине bsd строку

```
udpecho dgram udp wait jcs /usr/home/jcs/udpechod udpechod,
```

а в файл /etc/services – строку

```
udpecho  8001/udp
```

Затем переименуйте udpechol в udpechod и заставьте программу inetd перечитать свой конфигурационный файл. При запуске клиента udpclient на машине sparc получается:

```
sparc: $ udpclient bsd udpecho
one
28685: one
two
28686: two
three
28687: three
^C
sparc: $
```

Этот результат демонстрирует важную особенность UDP-серверов: они обычно не ведут диалог с клиентом. Иными словами, сервер получает один запрос

и посылает один ответ. Для UDP-серверов, запускаемых через inetd, типичными будут следующие действия: получить запрос, отправить ответ, выйти. Выходить нужно как можно скорее, поскольку inetd не будет ждать других запросов, направленных в порт этого сервера, пока тот не завершит сеанс.

Из предыдущей распечатки видно, что, хотя складывается впечатление, будто udpclient ведет с udpecho1 диалог, в действительности каждый раз вызывается новый экземпляр сервера. Конечно, это неэффективно, но важнее то, что сервер не запоминает информации о состоянии диалога. Для udpecho1 это несущественно, так как каждое сообщение – это, по сути, отдельная транзакция. Но так бывает не всегда. Один из способов решения этой проблемы таков: сервер принимает сообщение от клиента (чтобы избежать бесконечного цикла), затем соединяется с ним, получая тем самым новый (эфемерный) порт, создает новый процесс и завершает работу. Диалог с клиентом продолжает созданный вновь процесс.

Примечание *Есть и другие возможности. Например, сервер мог бы обслуживать нескольких клиентов. Принимая датаграммы от нескольких клиентов, сервер амортизирует накладные расходы на свой запуск и не завершает сеанс, пока не обнаружит, что долго простаивает без дела. Преимущество этого метода в некотором упрощении клиентов за счет усложнения сервера.*

Чтобы понять, как это работает, внесите в код udpecho1 изменения, представленные в листинге 3.6.

Листинг 3.6. Вторая версия udpechod

udpecho2.c

```
#include "etcp.h"
int main( int argc, char **argv )
{
   struct sockaddr_in peer;
   int s;
   int rc;
   int len;
   int pidsz;
   char buf[ 120 ];

   pidsz = sprintf( buf, "%d: ", getpid() );
   len = sizeof( peer );
   rc = recvfrom( 0, buf + pidsz, sizeof( buf ) - pidsz,
      0, ( struct sockaddr * )&peer, &len );
   if ( rc < 0 )
      exit( 1 );

   s = socket( AF_INET, SOCK_DGRAM, 0 );
   if ( s < 0 )
      exit( 1 );
   if ( connect( s, ( struct sockaddr * )&peer, len ) < 0 )
      exit( 1 );
```

```
21     if ( fork() != 0 ) /* Ошибка или родительский процесс? */
22        exit( 0 );
23     /* Порожденный процесс. */
24     while ( strncmp( buf + pidsz, "done", 4 ) != 0 )
25     {
26        if ( write( s, buf, rc + pidsz ) < 0 )
27           break;
28        pidsz = sprintf( buf, "%d: ", getpid() );
29        alarm( 30 );
30        rc = read( s, buf + pidsz, sizeof( buf ) - pidsz );
31        alarm( 0 );
32        if ( rc < 0 )
33           break;
34     }
35     exit( 0 );
36 }
```

udpecho2.c

udpecho2

10-15 Получаем идентификатор процесса, записываем его в начало буфера и читаем первое сообщение так же, как в udpecho1.

16-20 Получаем новый сокет и подсоединяем его к клиенту, пользуясь адресом в структуре peer, которая была заполнена при вызове recvfrom.

21-22 Родительский процесс разветвляется и завершается. В этот момент inetd может возобновить прослушивание хорошо известного порта сервера в ожидании новых сообщений. Важно отметить, что потомок использует номер порта new, привязанный к сокету s в результате вызова connect.

24-35 Затем посылаем клиенту полученное от него сообщение, только с добавленным в начало идентификатором процесса. Продолжаем читать сообщения от клиента, добавлять к ним идентификатор процесса-потомка и отправлять их назад, пока не получим сообщение, начинающееся со строки done. В этот момент сервер завершает работу. Вызовы alarm, окружающие операцию чтения на строке 30, – это защита от клиента, который закончил сеанс, не послав done. В противном случае сервер мог бы «зависнуть» навсегда. Поскольку установлен обработчик сигнала SIGALRM, UNIX завершает программу при срабатывании таймера.

Переименовав новую версию исполняемой программы в udpechod и запустив ее, вы получили следующие результаты:

```
sparc: $ udpclient bsd udpecho
one
28743: one
two
28744: two
three
```

```
28744: three
done
^C
sparc: $
```

На этот раз, как видите, в первом сообщении пришел идентификатор родительского процесса (сервера, запущенного inetd), а в остальных – один и тот же идентификатор (потомка). Теперь вы понимаете, почему udpclient всякий раз извлекает адрес сервера: ему нужно знать новый номер порта (а возможно, и новый IP-адрес, если сервер работает на машине с несколькими сетевыми интерфейсами), в который посылать следующее сообщение. Разумеется, это необходимо делать только для первого вызова recvfrom, но для упрощения здесь не выделяется особый случай.

Резюме

В этом разделе показано, как заставить приложение работать в сети, приложив совсем немного усилий. Демон inetd берет на себя ожидание соединений или датаграмм, дублирует дескриптор сокета на stdin, stdout и stderr и запускает приложение. После этого приложение может просто читать из stdin или писать в stdout либо stderr, не имея информации о том, что оно работает в сети. Рассмотрен пример простого фильтра, в котором вообще нет кода, имеющего отношение к сети. Но этот фильтр тем не менее прекрасно работает в качестве сетевого сервиса, если запустить его через inetd.

Здесь также приведен пример UDP-сервера, который способен вести продолжительный диалог с клиентами. Для этого серверу пришлось получить новый сокет и номер порта, а затем создать новый процесс и выйти.

Совет 18. Подумайте о том, чтобы хорошо известный номер порта назначался вашему серверу с помощью tcpmux

Проектировщик сетевого сервера сталкивается с проблемой выбора номера для хорошо известного порта. Агентство по выделению имен и уникальных параметров протоколов Internet (Internet Assigned Numbers Authority – IANA) подразделяет все номера портов на три группы: «официальные» (хорошо известные), зарегистрированные и динамические, или частные.

Примечание *Термин «хорошо известный порт» используется в общем смысле – как номер порта доступа к серверу. Строго говоря, хорошо известные порты контролируются агентством IANA.*

Хорошо известные – это номера портов в диапазоне от 0 до 1023. Они контролируются агентством IANA. Зарегистрированные номера портов находятся в диапазоне от 1024 до 49151. IANA не контролирует их, но регистрирует и публикует в качестве услуги сетевому сообществу. Динамические или частные порты имеют номера от 49152 до 65535. Предполагается, что эти порты будут использоваться

как эфемерные, но многие системы не следуют этому соглашению. Так, системы, производные от BSD, традиционно выбирают номера эфемерных портов из диапазона от 1024–5000. Полный список всех присвоенных IANA и зарегистрированных номеров портов можно найти на сайте http://www.isi.edu/in-notes/iana/assignment/port-numbers/.

Проектировщик сервера может получить от IANA зарегистрированный номер порта.

Примечание *Чтобы подать заявку на получение хорошо известного или зарегистрированного номера порта, зайдите на Web-страницу http://www.isi.edu/cgi-bin/iana/port-numbers.pl.*

Это, конечно, не помешает другим использовать тот же номер, и рано или поздно два сервера, работающие с одним и тем же номером порта, окажутся на одной машине. Обычно эту проблему решают, выделяя некоторый номер порта по умолчанию, но позволяя задать другой в командной строке.

Другое более гибкое решение, но применяемое реже, состоит в том, чтобы использовать возможность inetd (совет 17), которая называется мультиплексором портов TCP (TCP Port Service Multiplexor – TCPMUX). Сервис TCPMUX описан в RFC 1078 [Lotter 1988]. Мультиплексор прослушивает порт 1 в ожидании TCP-соединений. Клиент соединяется с TCPMUX и посылает ему строку с именем сервиса, который он хочет запустить. Строка должна завершаться символами возврата каретки и перевода строки (<CR><LF>). Сервер или, возможно, TCPMUX посылает клиенту один символ: + (подтверждение) или – (отказ), за которым следует необязательное пояснительное сообщение, завершаемое последовательностью <CR><LF>. Имена сервисов (без учета регистра) также хранятся в файле inetd.conf, но начинаются со строки tcpmux/, чтобы отличить их от обычных сервисов. Если имя сервиса начинаются со знака +, то подтверждение посылает TCPMUX, а не сервер. Это позволяет таким серверам, как rlnumd (листинг 3.3), которые проектировались без учета TCPMUX, все же воспользоваться предоставляемым им сервисом.

Например, если вы захотите запустить сервис подсчета строк из совета 17 в качестве TCPMUX-сервера, то надо добавить в файл inetd.conf строку

```
tcpmux/+rlnumd stream tcp nowait jcs /usr/jome/jcs/rlnumd rlnumd
```

Для тестирования заставьте inetd перечитать свой конфигурационный файл, а затем подсоединитесь к нему с помощью telnet, указав имя сервиса TCPMUX:

```
bsd: $ telnet localhost tcpmux
Trying 127.0.0.1 ...
Connected to localhost
Escape character is "^]".
rlnumd
+Go
hello
   1: hello
```

```
world
  2: world
^]
telnet> quit
Connection closed
bsd: $
```

К сожалению, сервис TCPMUX поддерживается не всеми операционными системами и даже не всеми UNIX-системами. Но, с другой стороны, его реализация настолько проста, что возможно написать собственную версию. Поскольку TCPMUX должен делать почти то же, что и `inetd` (за исключением мониторинга нескольких сокетов) заодно будут проиллюстрированы те идеи, которые лежат в основе `inetd`. Начнем с определения констант, глобальных переменных и функции `main` (листинг 3.7).

Листинг 3.7. tcpmux – константы, глобальные переменные и main

tcpmux.c

```
#include "etcp.h"

#define MAXARGS   10     /* Максимальное число аргументов сервера. */
#define MAXLINE   256    /* Максимальная длина строки в tcpmux.conf. */
#define NSERVTAB  10     /* Число элементов в таблице service_table. */
#define CONFIG "tcpmux.conf"

typedef struct
{
   int flag;
   char *service;
   char *path;
   char *args[ MAXARGS + 1 ];
    } servtab_t;

   int ls;              /* Прослушиваемый сокет. */
   servtab_t service_table[ NSERVTAB + 1 ];

   int main( int argc, char **argv )
   {
   struct sockaddr_in peer;
   int s;
   int peerlen;

   /* Инициализировать и запустить сервер tcpmux. */

   INIT();
   parsetab();
   switch ( argc )
   {
      case 1:      /* Все по умолчанию. */
         ls = tcp_server( NULL, "tcpmux" );
         break;

      case 2:      /* Задан интерфейс и номер порта. */
         ls = tcp_server( argv[ 1 ], "tcpmux" );
         break;
```

```
31        case 3:      /* Заданы все параметры. */
32           ls = tcp_server( argv[ 1 ], argv[ 2 ] );
33           break;

34        default:
35           error( 1, 0, "Вызов: %s [ интерфейс [ порт ] ]\n",
36              program_name );
37     }
38     daemon( 0, 0 );
39     signal( SIGCHLD, reaper );

40     /* Принять соединения с портом tcpmux. */

41     for ( ;; )
42     {
43        peerlen = sizeof( peer );
44        s = accept( ls, ( struct sockaddr * )&peer, &peerlen );
45        if ( s < 0 )
46           continue;
47        start_server( s );
48        CLOSE( s );
49     }
50  }
```

tcpmux.c

main

6-12 Структура servtab_t определяет тип элементов в таблице service_table. Поле flag устанавливается в TRUE, если подтверждение должен посылать tcpmux, а не сам сервер.

22 В начале вызываем функцию parsetab, которая читает и разбирает файл tcpmux.conf и строит таблицу service_table. Текст процедуры parsetab приведен в листинге 3.9.

23-37 Данная версия tcpmux позволяет пользователю задать интерфейс или порт, который будет прослушиваться. Этот код инициализирует сервер с учетом заданных параметров, а остальным присваивает значения по умолчанию.

38 Вызываем функцию daemon, чтобы перевести процесс tcpmux в фоновый режим и разорвать его связь с терминалом.

39 Устанавливаем обработчик сигнала SIGCHLD. Это не дает запускаемым серверам превратиться в «зомби» (и зря расходовать системные ресурсы) при завершении.

Примечание *В некоторых системах функция signal – это интерфейс к сигналам со старой «ненадежной» семантикой. В этом случае надо пользоваться функцией sigaction, которая обеспечивает семантику надежных сигналов. Обычно эту проблему решают путем создания собственной функции signal, которая вызывает из себя sigaction. Такая реализация приведена в приложении 1.*

41-49 В этом цикле принимаются соединения с `tcpmux` и вызывается функция `start_server`, которая создает новый процесс с помощью `fork` и запускает запрошенный сервер с помощью exec.

Теперь надо познакомимся с функцией `start_server` (листинг 3.8). Именно здесь выполняются основные действия.

Листинг 3.8. Функция start_server

tcpmux.c

```
static void start_server( int s )
{
   char line[ MAXLINE ];
   servtab_t *stp;
   int rc;
   static char err1[] = "-не могу прочесть имя сервиса \r\n";
   static char err2[] = "-неизвестный сервис\r\n";
   static char err3[] = "-не могу запустить сервис\r\n";
   static char ok[] = "+OK\r\n";

   rc = fork();
   if ( rc < 0 )       /* Ошибка вызова fork. */
   {
      write( s, err3, sizeof( err3 ) - 1 );
      return;
   }
   if ( rc != 0 )      /* Родитель. */
      return;

   /* Процесс-потомок. */

   CLOSE( ls );    /* Закрыть прослушивающий сокет. */

   alarm( 10 );
   rc = readcrlf( s, line, sizeof( line ) );
   alarm( 0 );
   if ( rc <= 0 )
   {
      write( s, err1, sizeof( err1 ) - 1 );
      EXIT( 1 );
   }

   for ( stp = service_table; stp->service; stp++ )
      if ( strcasecmp( line, stp->service ) == 0 )
         break;
   if ( !stp->service )
   {
      write( s, err2, sizeof( err2 ) - 1 );
      EXIT( 1 );
   }

   if ( stp->flag )
      if ( write( s, ok, sizeof( ok ) - 1 ) < 0 )
```

```
38          EXIT( 1 );
39     dup2( s, 0 );
40     dup2( s, 1 );
41     dup2( s, 2 );
42     CLOSE( s );
43     execv( stp->path, stp->args );
44     write( 1, err3, sizeof( err3 ) - 1 );
45     EXIT( 1 );
46 }
```

tcpmux.c

start_server

10-17 Сначала с помощью системного вызова fork создаем новый процесс, идентичный своему родителю. Если fork завершился неудачно, то посылаем клиенту сообщение об ошибке и возвращаемся (раз fork не отработал, то процесса-потомка нет, и управление возвращается в функцию main родительского процесса). Если fork завершился нормально, то это родительский процесс, и управление возвращается.

19-27 В созданном процессе закрываем прослушивающий сокет и из подсоединенного сокета читаем имя сервиса, которому нужно запустить клиент. Окружаем операцию чтения вызовами alarm, чтобы завершить работу, если клиент так и не пришлет имя сервиса. Если функция readcrlf возвращает ошибку, посылаем клиенту сообщение и заканчиваем сеанс. Текст readcrlf приведен ниже в листинге 3.10.

28-35 Ищем в таблице service_table имя запрошенного сервиса. Если оно отсутствует, то посылаем клиенту сообщение об ошибке и завершаем работу.

36-38 Если имя сервиса начинается со знака +, посылаем клиенту подтверждение. В противном случае даем возможность сделать это серверу.

39-45 С помощью системного вызова dup дублируем дескриптор сокета на stdin, stdout и stderr, после чего закрываем исходный сокет. И, наконец, подменяем процесс процессом сервера с помощью вызова execv. После этого запрошенный клиентом сервер – это процесс-потомок. Если execv возвращает управление, то сообщаем клиенту, что не смогли запустить запрошенный сервер, и завершаем сеанс.

В листинге 3.9 приведен текст подпрограммы parsetab. Она выполняет простой, но несколько утомительный разбор файла tcpmux.conf. Файл имеет следующий формат:

```
имя_сервиса путь аргументы ...
```

Листинг 3.9. Функция parsetab

tcpmux.c

```
static void parsetab( void )
{
    FILE *fp;
    servtab_t *stp = service_table;
    char *cp;
```

```
int i;
int lineno;
char line[ MAXLINE ];

fp = fopen( CONFIG, "r" );
if ( fp == NULL )
   error( 1, errno, "не могу открыть %s", CONFIG );
lineno = 0;
while ( fgets( line, sizeof( line ), fp ) != NULL )
{
   lineno++;
   if ( line[ strlen( line ) - 1 ] != '\n' )
      error( 1, 0, "строка %d слишком длинная\n", lineno );
   if ( stp >= service_table + NSERVTAB )
      error( 1, 0, "слишком много строк в tcpmux.conf\n" );
   cp = strchr( line, '#' );
   if ( cp != NULL )
      *cp = '\0';
   cp = strtok( line, " \t\n" );
   if ( cp == NULL )
      continue;
   if ( *cp == '+' )
   {
      stp->flag = TRUE;
      cp++;
      if ( *cp == '\0' || strchr( " \t\n", *cp ) != NULL )
         error( 1, 0, "строка %d: пробел после '+'\n",
            lineno );
   }
   stp->service = strdup( cp );
   if ( stp->service == NULL )
      error( 1, 0, "не хватило памяти\n" );
   cp = strtok( NULL, " \t\n" );
   if ( cp == NULL )
      error( 1, 0, "строка %d: не задан путь (%s)\n",
         lineno, stp->service );
   stp->path = strdup( cp );
   if ( stp->path == NULL )
      error( 1, 0, "не хватило памяти\n" );
   for ( i = 0; i < MAXARGS; i++ )
   {
      cp = strtok( NULL, " \t\n" );
      if ( cp == NULL )
         break;
      stp->args[ i ] = strdup( cp );
      if ( stp->args[ i ] == NULL )
         error( 1, 0, "не хватило памяти\n" );
   }
   if ( i >= MAXARGS && strtok( NULL, " \t\n" ) != NULL )
   error( 1, 0, "строка %d: слишком много аргументов (%s)\n",
```

```
            lineno, stp->service );
        stp->args[ i ] = NULL;
        stp++;
    }
    stp->service = NULL;
    fclose ( fp );
}
```

tcpmux.c

Показанная в листинге 3.10 функция `readcrlf` читает из сокета по одному байту. Хотя это и неэффективно, но гарантирует, что будет прочитана только первая строка данных, полученных от клиента. Все данные, кроме первой строки, предназначены серверу. Если бы вы буферизовали ввод, а клиент послал бы больше одной строки, то часть данных, адресованных серверу, считывал бы `tcpmux`, и они были бы потеряны.

Обратите внимание, что `readcrlf` принимает также и строку, завершающуюся только символом новой строки. Это находится в полном соответствии с принципом устойчивости [Postel 1981a], который гласит: «Подходите не слишком строго к тому, что принимаете, но очень строго – к тому, что посылаете». В любом случае как <CR><LF>, так и одиночный <LF> отбрасываются.

Определение функции `readcrlf` такое же, как функций `read`, `readline`, `readn` и `readvrec`:

```
#include "etcp.h"

int readcrlf( SOCKET s, char *buf, size_t len );
```

Возвращаемое значение: число прочитанных байт или –1 в случае ошибки.

Листинг 3.10. Функция readcrlf

readcrlf.c

```
int readcrlf( SOCKET s, char *buf, size_t len )
{
    char *bufx = buf;
    int rc;
    char c;
    char lastc = 0;

    while ( len > 0 )
    {
        if ( ( rc = recv( s, &c, 1, 0 ) ) != 1 )
        {
            /*
             *  Если нас прервали, повторим,
             *  иначе вернем EOF или код ошибки.
             */

            if ( rc < 0 && errno == EINTR )
                continue;
```

```
            return rc;
        }
        if ( c == '\n' )
        {
            if ( lastc == '\r' )
                buf--;
            *buf = '\0';            /* Не включать <CR><LF>. */
            return buf - bufx;
        }

        *buf++ = c;
        lastc = c;
        len--;
    }
    set_errno( EMSGSIZE );
    return -1;
}
```

readcrlf.c

И наконец рассмотрим функцию `reaper` (листинг 3.11). Когда сервер, запущенный с помощью `tcpmux`, завершает сеанс, UNIX посылает родителю (то есть `tcpmux`) сигнал `SIGCHLD`. При этом вызывается обработчик сигнала `reaper`, который, в свою очередь, вызывает `waitpid` для получения статуса любого из завершившихся потомков. В системе UNIX это необходимо, поскольку процесс-потомок может возвращать родителю свой статус завершения (например, аргумент функции `exit`).

Примечание *В некоторых вариантах UNIX потомок возвращает и другую информацию. Так, в системах, производных от BSD, возвращается сводная информация о количестве ресурсов, потребленных завершившимся процессом и всеми его потомками. Во всех системах UNIX, по меньшей мере, возвращается указание на то, как завершился процесс: из-за вызова `exit` (передается также код возврата) или из-за прерывания сигналом (указывается номер сигнала).*

Пока родительский процесс не заберет информацию о завершении потомка с помощью вызова `wait` или `waitpid`, система UNIX должна удерживать ту часть ресурсов, занятых процессом-потомком, в которой хранится информация о состоянии. Потомки, которые уже завершились, но еще не передали родителю информацию о состоянии, называются мертвыми (defunct) или «зомби».

Листинг 3.11. Функция reaper

tcpmux.c

```
void reaper( int sig )
{
    int waitstatus;

    while ( waitpid( -1, &waitstatus, WNOHANG ) > 0 ) {;}
}
```

tcpmux.c

Протестируйте tcpmux, создав файл tcpmux.conf из одной строки:

```
+rlnum /usr/hone/jcs/rlnumd rlnumd
```

Затем запустите tcpmux на машине sparc, которая не поддерживает сервиса TCPMUX, и соединитесь с ним, запустив telnet на машине bsd.

```
sparc: # tcpmux

bsd: $ telnet sparc tcpmux
Trying 127.0.0.1 ...
Connected to sparc
Escape character is '^]'.
rlnumd
+OK
hello
  1: hello
world
  2: world
^]
telnet> quit
Connection closed
bsd: $
```

Резюме

Сервис TCPMUX, имеющийся на очень многих системах, помогает решить проблему выбора хорошо известного номера порта сервера. Здесь реализована собственная версия демона tcpmux, так что если в какой-то системе его нет, то им можно воспользоваться.

Совет 19. Подумайте об использовании двух TCP-соединений

Во многих приложениях удобно разрешить нескольким процессам или потокам читать из TCP-соединений и писать в них. Особенно распространена эта практика в системе UNIX, где по традиции создается процесс-потомок, который, например, пишет в TTY-соединение, тогда как родитель занимается чтением.

Типичная ситуация изображена на рис. 3.3, где показан эмулятор терминала. Родительский процесс большую часть времени блокирован в ожидании ввода из TTY-соединения. Когда на вход поступают данные, родитель читает их и выводит на экран, возможно, изменяя на ходу формат. Процесс-потомок в основном блокирован, ожидая ввода с клавиатуры. Когда пользователь вводит данные, потомок выполняет необходимые преобразования и записывает данные в TTY-соединение.

Эта стратегия удобна, поскольку позволяет автоматически мультиплексировать ввод с клавиатуры и из TTY-соединения и разнести логику преобразования кодов клавиш и форматирования вывода на экран по разным модулям. За счет этого программа получается концептуально проще, чем в случае нахождения кода

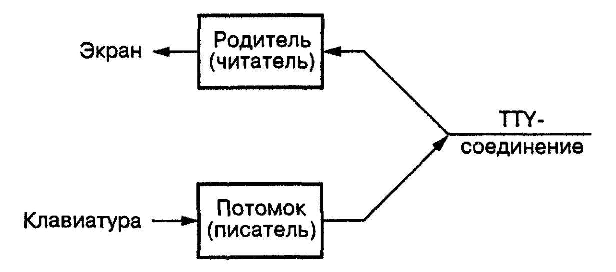


Рис. 3.3. Два процесса, обслуживающие TTY-соединение

в одном месте. В действительности, до появления в операционной системе механизма `select` это был единственный способ обработки поступления данных из нескольких источников.

Архитектура с одним соединением

Следует заметить, что ничего не изменится, если на рис. 3.3 вместо TTY-соединения будет написано TCP-соединение. Поэтому та же техника может применяться (и часто применяется) для работы с сетевыми соединениями. Кроме того, использование потоков вместо процессов почти не сказывается на ситуации, изображенной на рисунке, поэтому этот метод пригоден и для многопоточной среды.

Правда, есть одна трудность. Если речь идет о TTY-соединении, то ошибки при операции записи возвращаются самим вызовом `write`, тогда как в случае TCP ошибка, скорее всего, будет возвращена последующей операцией чтения (совет 15). В многопроцессной архитектуре процессу-читателю трудно уведомить процесс-писатель об ошибке. В частности, если приложение на другом конце завершается, то об этом узнает читатель, который должен как-то известить писателя.

Немного изменим точку зрения и представим себе приложение, которое принимает сообщения от внешней системы и посылает их назад по TCP-соединению. Сообщения передаются и принимаются асинхронно, то есть не для каждого входного сообщения генерируется ответ, и не каждое выходное сообщение посылается в ответ на входное. Допустим также, что сообщения нужно переформатировать на

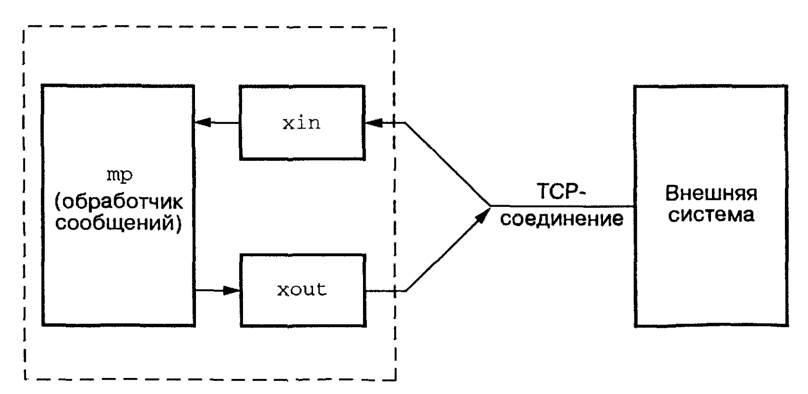


Рис. 3.4. Приложение, обменивающееся сообщениями по TCP-соединению

входе или выходе из приложения. Тогда представленная на рис. 3.4 архитектура многопроцессного приложения оказывается вполне разумной.

На этом рисунке процесс xin читает данные от внешней системы, накапливает их в очереди сообщений, переформатирует и передает главному процессу обработки сообщений. Аналогично процесс xout приводит выходное сообщение к формату, требуемому внешней системой, и записывает данные в TCP-соединение. Главный процесс mp обрабатывает отформатированные входные сообщения и генерирует выходные сообщения. Оставляем неспецифицированным механизм межпроцессного взаимодействия (IPC) между тремя процессами. Это может быть конвейер, разделяемая память, очереди сообщений или еще что-то. Подробнее все возможности рассмотрены в книге [Stevens 1999]. В качестве реального примера такого рода приложения можно было бы привести шлюз, через который передаются сообщения между системами. Причем одна из систем работает по протоколу TCP, а другая – по какому-либо иному протоколу.

Если обобщить этот пример, учитывая дополнительные внешние системы с иными требованиями к формату сообщений, то становится ясно, насколько гибкие возможности предоставляет описанный метод. Для каждого внешнего хоста имеется свой набор коммуникационных процессов, работающих только с его сообщениями. Такая система концептуально проста, позволяет вносить изменения, относящиеся к одному из внешних хостов, не затрагивая других, и легко конфигурируется для заданного набора внешних хостов – достаточно лишь запустить свои коммуникационные процессы для каждого хоста.

Однако при этом остается нерешенной вышеупомянутая проблема: процесс-писатель не может получить сообщение об ошибке после операции записи. А иногда у приложения должна быть точная информация о том, что внешняя система действительно получила сообщение, и необходимо организовать протокол подтверждений по типу того, что обсуждался в совете 9. Это означает, что нужно либо создать отдельный коммуникационный канал между процессами xin и xout, либо xin должен посылать информацию об успешном получении и об ошибках процессу mp, который, в свою очередь, переправляет их процессу xout. То и другое усложняет взаимодействие процессов.

Можно, конечно, отказаться от многопроцессной архитектуры и оставить всего один процесс, добавив select для мультиплексирования сообщений. Однако при этом приходится жертвовать гибкостью и концептуальной простотой.

Далее в этом разделе рассмотрим альтернативную архитектуру, при которой сохраняется гибкость, свойственная схеме на рис. 3.4, но каждый процесс самостоятельно следит за своим TCP-соединением.

Архитектура с двумя соединениями

Процессы xin и xout на рис. 3.4 делят между собой единственное соединение с внешней системой, но возникают трудности при организации разделения информации о состоянии этого соединения. Кроме того, с точки зрения каждого из процессов xin и xout, это соединение симплексное, то есть данные передаются по нему только в одном направлении. Если бы это было не так, то xout «похищал» бы входные данные у xin, а xin мог бы исказить данные, посылаемые xout.

Решение состоит в том, чтобы завести два соединения с внешней системой – по одному для xin и xout. Полученная после такого изменения архитектура изображена на рис. 3.5.

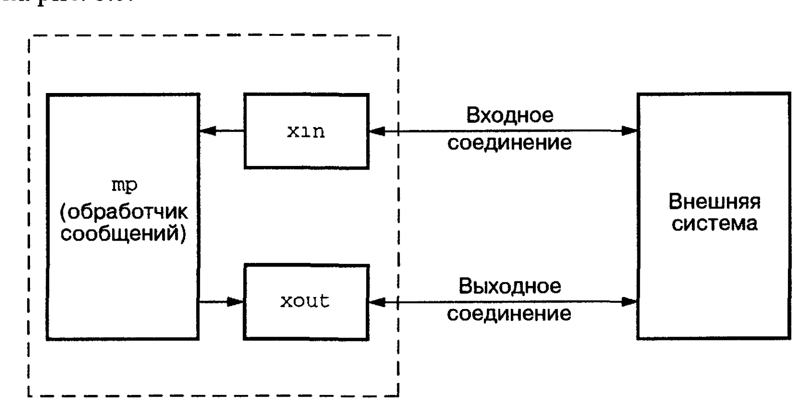


Рис. 3.5. Приложение, обменивающееся сообщениями по двум TCP-соединениям

Если система не требует отправки подтверждений на прикладном уровне, то при такой архитектуре выигрывает процесс xout, который теперь имеет возможность самостоятельно узнавать об ошибках и признаке конца файла, посланных партнером. С другой стороны, xout становится немного сложнее, поскольку для получения уведомления об этих событиях он должен выполнять операцию чтения. К счастью, это легко можно обойти с помощью вызова select.

Чтобы это проверить, запрограммируем простой процесс xout, который читает данные из стандартного ввода и записывает их в TCP-соединение. Программа, показанная в листинге 3.12, с помощью вызова select ожидает поступления данных из соединения, хотя реально может прийти только EOF или извещение об ошибке.

Листинг 3.12. Программа, готовая к чтению признака конца файла или ошибки

xout1.c

```
#include "etcp.h"

int main( int argc, char **argv )
{
    fd_set allreads;
    fd_set readmask;
    SOCKET s;
    int rc;
    char buf[ 128 ];

    INIT();
    s = tcp_client( argv[ 1 ], argv[ 2 ] );
    FD_ZERO( &allreads );
    FD_SET( s, &allreads );
    FD_SET( 0, &allreads );
    for ( ;; )
    {
        readmask = allreads;
```

```
17         rc = select( s + 1, &readmask, NULL, NULL, NULL );
18         if ( rc <= 0 )
19            error( 1, rc ? errno : 0, "select вернул %d", rc );
20         if ( FD_ISSET( 0, &readmask ) )
21         {
22            rc = read( 0, buf, sizeof( buf ) - 1 );
23            if ( rc < 0 )
24               error( 1, errno, "ошибка вызова read" );
25            if ( send( s, buf, rc, 0 ) < 0 )
26               error( 1, errno, "ошибка вызова send" );
27         }
28         if ( FD_ISSET( s, &readmask ) )
29         {
30            rc = recv( s, buf, sizeof( buf ) - 1, 0 );
31            if ( rc == 0 )
32               error( 1, 0, "сервер отсоединился\n" );
33            else if ( rc < 0 )
34               error( 1, errno, "ошибка вызова recv" );
35            else
36            {
37               buf[ rc ] = '\0';
38               error( 1, 0, "неожиданный вход [%s]\n", buf );
39            }
40         }
41     }
42 }
```

xout1.c

Инициализация

9-13 Выполняем обычную инициализацию, вызываем функцию `tcp_client` для установки соединения и готовим `select` для извещения о наличии входных данных в стандартном вводе или в только что установленном TCP-соединении.

Обработка событий stdin

20-27 Если данные пришли из стандартного ввода, посылаем их удаленному хосту через TCP-соединение.

Обработка событий сокета

28-40 Если пришло извещение о наличии доступных для чтения данных в сокете, то проверяем, это EOF или ошибка. Никаких данных по этому соединению не должно быть получено, поэтому если пришло что-то иное, то печатаем диагностическое сообщение и завершаем работу.

Продемонстрировать работу `xout1` можно, воспользовавшись программой `keep` (листинг 2.30) в качестве внешней системы и простым сценарием на языке интерпретатора команд shell для обработки сообщений (`mp` на рис. 3.5). Этот сценарий каждую секунду выводит на `stdout` слово `message` и счетчик.

```
MSGNO=1
while true
```

```
do
   echo message $MSGNO
   sleep 1
   MSGNO="expr $MSGNO + 1"
done
```

Обратите внимание, что в этом случае xout1 использует конвейер в качестве механизма IPC. Поэтому в таком виде программа xout1 не переносится на платформу Windows, поскольку вызов select работает под Windows только для сокетов. Можно было бы реализовать взаимодействие между процессами с помощью TCP или UDP, но тогда потребовался бы более сложный обработчик сообщений.

Для тестирования xout1 запустим сначала «внешнюю систему» в одном окне, а обработчик сообщений и xout1 – в другом.

```
bsd: $ keep 9000                    | bsd: $ mp | xout1 localhost 9000
message 1                           |
message 2                           |
message 3                           |
message 4                           |
^C "Внешняя система"                | xout1: сервер отсоединился
   завершила работу                 | Broken pipe
bsd: $                              | bsd: $
```

Сообщение Broken pipe напечатал сценарий mp. При завершении программы xout1 конвейер между ней и сценарием закрывается. Когда сценарий пытается записать в него следующую строку, происходит ошибка, и сценарий завершается с сообщением Broken pipe.

Более интересна ситуация, когда между внешней системой и приложением, обрабатывающим сообщения, необходим обмен подтверждениями. В этом случае придется изменить и xin, и xout (предполагая, что подтверждения нужны в обоих направлениях; если нужно только подтверждать внешней системе прием сообщений, то изменения надо внести лишь в xin). Разработаем пример только процесса-писателя (xout). Изменения в xin аналогичны.

Новый процесс-писатель обязан решать те же проблемы, с которыми вы столкнулись при обсуждении пульсаций в совете 10. После отправки сообщения удаленный хост должен прислать нам подтверждение до того, как сработает таймер. Если истекает тайм-аут, необходима какая-то процедура восстановления после ошибки. В примере работа просто завершается.

При разработке нового «писателя» xout2 вы не будете принимать сообщений из стандартного ввода, пока не получите подтверждения от внешней системы о том, что ей доставлено последнее ваше сообщение. Возможен и более изощренный подход с использованием механизма тайм-аутов, описанного в совете 20. Далее он будет рассмотрен, но для многих систем вполне достаточно той простой схемы, которую будет применена. Текст xout2 приведен в листинге 3.13.

Листинг 3.13. Программа, обрабатывающая подтверждения

xout2.c

```
#include "etcp.h"
#define ACK    0x6/* Символ подтверждения ACK. */
```

```
int main( int argc, char **argv )
{
   fd_set allreads;
   fd_set readmask;
   fd_set sockonly;
   struct timeval tv;
   struct timeval *tvp = NULL;
   SOCKET s;
   int rc;
   char buf[ 128 ];
   const static struct timeval T0 = { 2, 0 };

   INIT();
   s = tcp_client( argv[ 1 ], argv[ 2 ] );
   FD_ZERO( &allreads );
   FD_SET( s, &allreads );
   sockonly = allreads;
   FD_SET( 0, &allreads );
   readmask = allreads;
   for ( ;; )
   {
      rc = select( s + 1, &readmask, NULL, NULL, tvp );
      if ( rc < 0 )
         error( 1, errno, "ошибка вызова select" );
      if ( rc == 0 )
         error( 1, 0, "тайм-аут при приеме сообщения\n" );

      if ( FD_ISSET( s, &readmask ) )
      {
         rc = recv( s, buf, sizeof( buf ), 0 );
         if ( rc == 0 )
            error( 1, 0, "сервер отсоединился\n" );
         else if ( rc < 0 )
            error( 1, errno, "ошибка вызова recv" );
         else if ( rc != 1 || buf[ 0 ] != ACK )
            error( 1, 0, "неожиданный вход [%c]\n", buf[ 0 ] );
         tvp = NULL;  /* Отключить таймер */
         readmask = allreads; /* и продолжить чтение из stdin. */
      }

      if ( FD_ISSET( 0, &readmask ) )
      {
      rc = read( 0, buf, sizeof( buf ) );
      if ( rc < 0 )
         error( 1, errno, "ошибка вызова read" );
      if ( send( s, buf, rc, 0 ) < 0 )
         error( 1, errno, "ошибка вызова send" );
      tv = T0;                /* Переустановить таймер. */
      tvp = &tv;              /* Взвести таймер */
      readmask = sockonly; /* и прекратить чтение из stdin. */
   }
  }
}
```

— xout2.c

Инициализация

14-15 Стандартная инициализация TCP-клиента.

16-20 Готовим две маски для select: одну для приема событий из stdin и TCP-сокета, другую для приема только событий из сокета. Вторая маска sockonly применяется после отправки данных, чтобы не читать новые данные из stdin, пока не придет подтверждение.

Обработка событий таймера

26-27 Если при вызове select произошел тайм-аут (не получено вовремя подтверждение), то печатаем диагностическое сообщение и завершаем сеанс.

Обработка событий сокета

28-39 Если пришло извещение о наличии доступных для чтения данных в сокете, проверяем, это EOF или ошибка. Если да, то завершаем работу так же, как в листинге 3.12. Если получены данные, убеждаемся, что это всего один символ ACK. Тогда последнее сообщение подтверждено, поэтому сбрасываем таймер, устанавливая переменную tvp в NULL, и разрешаем чтение из стандартного ввода, устанавливая маску readmask так, чтобы проверялись и сокет, и stdin.

Обработка событий в stdin

40-66 Получив событие stdin, проверяем, не признак ли это конца файла. Если чтение завершилось успешно, записываем данные в TCP-соединение.

47-50 Поскольку данные только что переданы внешней системе, ожидается подтверждение. Взводим таймер, устанавливая поля структуры tv и направляя на нее указатель tvp. В конце запрещаем события stdin, записывая в переменную readmask маску sockonly.

Для тестирования программы xout2 следует добавить две строки

```
if ( send( s1, "\006", 1, 0 ) < 0 )    /* \006 == ACK */
    error( 1, errno, "ошибка вызова send" );
```

перед записью на строке 24 в исходном тексте keep.c (листинг 2.30). Если выполнить те же действия, как и для программы xout1, то получим тот же результат с тем отличием, что xout2 завершает сеанс, не получив подтверждения от удаленного хоста.

Резюме

В этом разделе обсуждалась идея об использовании двух соединений между приложениями. Это позволяет отслеживать состояние соединения даже тогда, когда чтение и запись производятся в разных процессах.

Совет 20. Подумайте, не сделать ли приложение событийно-управляемым (1)

В этом и следующем разделах будет рассказано об использовании техники событийной управляемости в программировании TCP/IP. Будет разработан универсальный механизм тайм-аутов, позволяющий указать программе, что некоторое

событие должно произойти до истечения определенного времени, и асинхронно приступить к обработке этого события в указанное время. Здесь рассмотрим реализацию механизма таймеров, а в совете 21 вернемся к архитектуре с двумя соединениями и применим его на практике.

Разница между событийно-управляемым и обычным приложением хорошо иллюстрируется двумя написанными ранее программами: `hb_client2` (листинги 2.26 и 2.27) и `tcprw` (листинг 2.21). В `tcprw` поток управления последовательный: сначала из стандартного ввода читается строка и передается удаленному хосту, а затем от него принимается ответ и записывается на стандартный вывод. Обратите внимание, что нет возможности ничего принять от удаленного хоста, пока ожидается ввод из `stdin`. Как вы видели, в результате можно не знать, что партнер завершил сеанс и послал EOF. Ожидая также ответа от удаленного хоста, вы не можете читать новые данные из `stdin`. Это значит, что приложение, с точки зрения пользователя, слишком медленно реагирует. Кроме того, оно может «зависнуть», если удаленный хост «падает» до того, как приложение ответило.

Сравните это поведение с работой клиента `hb_client2`, который в любой момент способен принимать данные по любому соединению или завершиться по тайм-ауту. Ни одно из этих событий не зависит от другого, именно поэтому такая архитектура называется *событийно-управляемой.*

Заметим, что клиента `hb_client2` можно легко обобщить на большее число соединений или источников входной информации. Для этого существует механизм `select`, который позволяет блокировать процесс в ожидании сразу нескольких событий и возвращать ему управление, как только произойдет *любое* из них. В системе UNIX этот механизм, а также родственный ему вызов `poll`, имеющийся в системах на базе SysV, – это единственный эффективный способ обработки асинхронных событий в немногопоточной среде.

Примечание *До недавнего времени считалось, что из соображений переносимости следует использовать* `select`, *а не* `poll`, *так как на платформе Windows, а равно в современных UNIX-системах поддерживается именно* `select`, *тогда как* `poll` *встречается обычно в реализациях на базе SysV. Однако некоторые большие серверные приложения (например, Web-серверы), поддерживающие очень много одновременных соединений, применяют механизм* `poll`, *так как он лучше масштабируется на большое число дескрипторов. Дело в том, что* `select` *ограничен фиксированным числом дескрипторов. Обычно их не больше 1024, но бывает и меньше. Так, в системе FreeBSD и производных от нее по умолчанию предел равен 256. Для изменения значения по умолчанию нужно пересобирать ядро, что неудобно, хотя и возможно. Но и пересборка ядра лишь увеличивает предел, а не снимает его. Механизм же* `poll` *не имеет встроенных ограничений на число дескрипторов. Следует также принимать во внимание эффективность. Типичная реализация* `select` *может быть очень неэффективной при большом числе дескрипторов. Подробнее это рассматривается в работе [Banga and Mogul 1998]. (В этой работе приводится*

еще один пример возникновения трудностей при экстраполяции результатов, полученных в локальной сети, на глобальную. Эта тема обсуждалась в совете 12.) Проблема большого числа дескрипторов стоит особенно остро, когда ожидается немного событий на многих дескрипторах, то есть первый аргумент – `maxfd` *– велик, но с помощью* `FD_SET` *было зарегистрировано всего несколько дескрипторов. Это связано с тем, что ядро должно проверить все возможные дескрипторы (0, ..., maxfd), чтобы понять, ожидаются ли приложением события хотя бы на одном из них. В вызове* `poll` *используется массив дескрипторов, с помощью которого ядру сообщается о том, в каких событиях заинтересовано приложение, так что этой проблемы не возникает.*

Итак, использование `select` или `poll` позволяет мультиплексировать несколько событий ввода/вывода. Сложнее обстоит дело с несколькими таймерами, поскольку в вызове можно указать лишь одно значение тайм-аута. Чтобы решить эту проблему и создать тем самым более гибкое окружение для событийно-управляемых программ, следует разработать вариант вызова `select` – `tselect`. Хотя функции `timeout` и `untimeout`, связанные с `tselect`, построены по той же схеме, что и одноименные подпрограммы ядра UNIX, они работают в адресном пространстве пользователя и используют `select` для мультиплексирования ввода/вывода и получения таймера.

Таким образом, существуют три функции, ассоциированные с `tselect`. Прежде всего это сама `tselect`, которая применяется аналогично `select` для мультиплексирования ввода/вывода. Единственное отличие в том, что у `tselect` нет параметра `timeout` (это пятый параметр `select`). События таймера задаются с помощью вызова функции `timeout`, которая позволяет указать длительность таймера и действие, которое следует предпринять при его срабатывании. Вызов `untimeout` отменяет таймер до срабатывания.

Порядок вызова этих функций описан следующим образом:

```
#include "etcp.h"

int tselect( int maxfd, fd_set *rdmask, fd_set *wrmask, fd_set *exmask );
```

Возвращаемое значение: число готовых событий, 0 – если событий нет, –1 – ошибка.

```
unsigned int timeout( void ( *handler )( void * ), void *arg, int ms );
```

Возвращаемое значение: идентификатор таймера для передачи `untimeout`

```
void untimeout( unsigned int timerid );
```

Когда срабатывает таймер, ассоциированный с вызовом `timeout`, вызывается функция, заданная параметром *handler*, которой передается аргумент, заданный параметром *arg*. Таким образом, чтобы организовать вызов функции `retransmit` через полторы секунды с целым аргументом *sock*, нужно сначала написать

```
timeout( retransmit, ( void * ) sock, 1500 );
```

а затем вызывать `tselect`. Величина тайм-аута `ms` задается в миллисекундах, но надо понимать, что разрешающая способность системных часов может быть ниже. Для UNIX-систем типичное значение составляет 10 мс, поэтому не следует ожидать от таймера более высокой точности.

Примеры использования `tselect` будут приведены далее, а пока рассмотрим ее реализацию. В листинге 3.14 приведено определение структуры `tevent_t` и объявления глобальных переменных.

Листинг 3.14. Глобальные данные для tselect

tselect.c

```
 1 #include "etcp.h"

 2 #define NTIMERS 25

 3 typedef struct tevent_t tevent_t;
 4 struct tevent_t
 5 {
 6    tevent_t *next;
 7    struct timeval tv;
 8    void ( *func )( void * );
 9    void *arg;
10    unsigned int id;
11 };

12 static tevent_t *active = NULL;      /* Активные таймеры. */
13 static tevent_t *free_list = NULL;   /* Неактивные таймеры. */
```

tselect.c

Объявления

2 Константа `NTIMERS` определяет, сколько таймеров выделять за один раз. Сначала таймеров нет вовсе, поэтому при первом обращении к `timeout` будет выделено `NTIMERS` таймеров. Если все они задействованы и происходит очередное обращение к `timeout`, то выделяется еще `NTIMERS` таймеров.

3-11 Каждый таймер представляет отдельную структуру типа `tevent_t`. Структуры связаны в список полем `next`. В поле `tv` хранится время срабатывания таймера. Поля `func` и `arg` предназначены для хранения указателя на функцию обработки события таймера (которая вызывается при срабатывании) и ее аргумента. Наконец, идентификатор активного таймера хранится в поле `id`.

12 Порядок расположения активных таймеров в списке определяется моментом срабатывания. Глобальная переменная `active` указывает на первый таймер в списке.

13 Неактивные таймеры находятся в списке свободных. Когда функции `timeout` нужно получить новый таймер, она берет его из этого списка. Глобальная переменная `free_list` указывает на начало списка свободных.

Далее изучим функцию timeout и подпрограммы выделения таймеров (листинг 3.15).

Листинг 3.15. Функции timeout и allocate_timer

tselect.c

```
static tevent_t *allocate_timer( void )
{
      tevent_t *tp;

   if ( free_list == NULL ) /* нужен новый блок таймеров? */
   {
      free_list = malloc( NTIMERS * sizeof( tevent_t ) );
      if ( free_list == NULL )
         error( 1, 0, "не удалось получить таймеры\n" );
      for ( tp = free_list;
           tp < free_list + NTIMERS - 1; tp++ )
         tp->next = tp + 1;
      tp->next = NULL;
   }
   tp = free_list; /* Выделить первый. */
   free_list = tp->next; /* Убрать его из списка. */
   return tp;
   }

unsigned int timeout( void ( *func )( void * ), void *arg, int ms )
   {
   tevent_t *tp;
   tevent_t *tcur;
   tevent_t **tprev;
   static unsigned int id = 1; /* Идентификатор таймера. */

   tp = allocate_timer();
   tp->func = func;
   tp->arg = arg;
   if ( gettimeofday( &tp->tv, NULL ) < 0 )
      error( 1, errno, "timeout: ошибка вызова gettimeofday" );
   tp->tv.tv_usec += ms * 1000;
   if ( tp->tv.tv_usec > 1000000 )
   {
      tp->tv.tv_sec += tp->tv.tv_usec / 1000000;
      tp->tv.tv_usec %= 1000000;
   }
   for ( tprev = &active, tcur = active;
         tcur && !timercmp( &tp->tv, &tcur->tv, < ); /* XXX */
         tprev = &tcur->next, tcur = tcur->next )
   { ; }
   *tprev = tp;
   tp->next = tcur;
   tp->id = id++;   /* Присвоить значение идентификатору таймера. */
   return tp->id;
}
```

tselect.c

allocate_timer

4-13 Функция `allocate_timer` вызывается из `timeout` для получения свободного таймера. Если список свободных пуст, то из кучи выделяется память для NTIMERS структур `tevent_t`, и эти структуры связываются в список.

14-16 Выбираем первый свободный таймер из списка и возвращаем его вызывающей программе.

timeout

24-26 Получаем таймер и помещаем в поля `func` и `arg` значения переданных нам параметров.

27-34 Вычисляем момент срабатывания таймера, прибавляя значение параметра `ms` к текущему времени. Сохраняем результат в поле `tv`.

35-38 Ищем в списке активных место для вставки нового таймера. Вставить таймер нужно так, чтобы моменты срабатывания всех предшествующих таймеров были меньше либо равны, а моменты срабатывания всех последующих – больше момента срабатывания нового. На рис. 3.6 показан

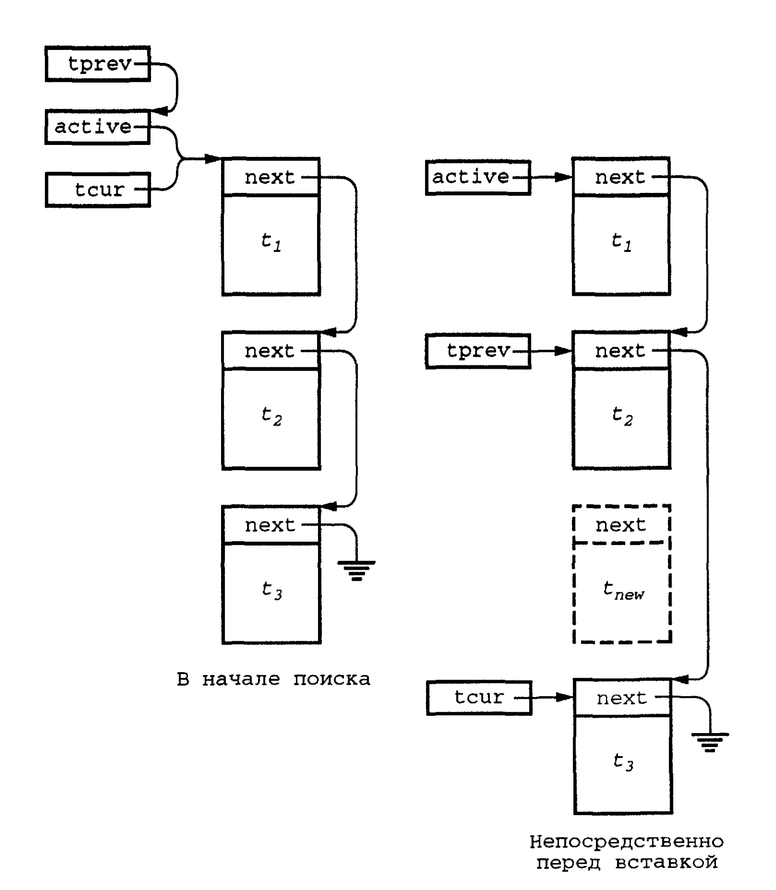


Рис. 3.6. Список активных таймеров до и после поиска точки вставки

процесс поиска и значения переменных `tcur` и `tprev`. Вставляем новый таймер так, что его момент срабатывания t_{new} удовлетворяет условию $t_0 \le t_1 \le t_{new} < t_2$. Обведенный курсивом прямоугольник t_{new} показывает позицию в списке, куда будет помещен новый таймер. Несколько странное использование макроса `timercmp` в строке 36 связано с тем, что версия в файле `winsock2.h` некорректна и не поддерживает оператора >=

27-34 Вставляем новый таймер в нужное место, присваиваем ему идентификатор и возвращаем этот идентификатор вызывающей программе. Возвращается идентификатор, а не адрес структуры `tevent_t`, чтобы избежать «гонки» (race condition). Когда таймер срабатывает, структура `tevent_t` возвращается в начало списка свободных. При выделении нового таймера будет использована именно эта структура. Если приложение теперь попытается отменить первый таймер, то при условии, что возвращается адрес структуры, а не индекс, будет отменен второй таймер. Эту проблему решает возврат идентификатора.

Идентификатор таймера, возвращенный в конце функции из листинга 3.15, используется функцией `untimeout` (листинг 3.16).

Листинг 3.16. Функция untimeout

```
—————————————————————————————————————————————————tselect.c
 1 void untimeout( unsigned int id )
 2 {
 3   tevent_t **tprev;
 4   tevent_t *tcur;
 5   for ( tprev = &active, tcur = active;
 6         tcur && id != tcur->id;
 7         tprev = &tcur->next, tcur = tcur->next )
 8   { ; }
 9   if ( tcur == NULL )
10   {
11     error( 0, 0,
12     "при вызове untimeout указан несуществующий таймер (%d)\n", id );
13     return;
14   }
15   *tprev = tcur->next;
16   tcur->next = free_list;
17   free_list = tcur;
18 }
—————————————————————————————————————————————————tselect.c
```

Поиск таймера

5-8 Ищем в списке активных таймер с идентификатором `id`. Этот цикл похож на тот, что используется в `timeout` (листинг 3.15).

9-14 Если в списке нет таймера, который пытаемся отменить, то выводим диагностическое сообщение и выходим.

Отмена таймера

15-17 Для отмены таймера исключаем структуру tevent_t из списка активных и возвращаем в список свободных.

Последняя из функций, работающих с таймерами, – это tselect (листинг 3.17)

Листинг 3.17. Функция tselect

tselect.c

```
int tselect( int maxp1, fd_set *re, fd_set *we, fd_set *ee )
{
   fd_set rmask;
   fd_set wmask;
   fd_set emask;
   struct timeval now;
   struct timeval tv;
   struct timeval *tvp;
   tevent_t *tp;
   int n;

   if ( re )
      rmask = *re;
   if ( we )
      wmask = *we;
   if ( ee )
      emask = *ee;
   for ( ;; )
   {
      if ( gettimeofday( &now, NULL ) < 0 )
        error( 1, errno, "tselect: ошибка вызова gettimeofday" );
      while ( active && !timercmp( &now, &active->tv, < ) )
      {
        active->func( active->arg );
        tp = active;
        active = active->next;
        tp->next = free_list;
        free_list = tp;
      }
      if ( active )
      {
        tv.tv_sec = active->tv.tv_sec - now.tv_sec;;
        tv.tv_usec = active->tv.tv_usec - now.tv_usec;
        if ( tv.tv_usec < 0 )
        {
            tv.tv_usec += 1000000;
            tv.tv_sec--;
        }
        tvp = &tv;
      }
```

```
40       else if ( re == NULL && we == NULL && ee == NULL )
41          return 0;
42       else
43          tvp = NULL;
44       n = select( maxp1, re, we, ee, tvp );
45       if ( n < 0 )
46          return -1;
47       if ( n > 0 )
48          return n;
49       if ( re )
50          *re = rmask;
51       if ( we )
52          *we = wmask;
53       if ( ee )
54          *ee = emask;
55    }
56 }
```

tselect.c

Сохранение масок событий

11-16 Поскольку при одном обращении к `tselect` может несколько раз вызываться `select`, сохраняем маски событий, передаваемых `select`.

Диспетчеризация событий таймера

19-28 Хотя в первой структуре `tevent_t`, находящейся в списке активных таймеров, время срабатывания меньше или равно текущему времени, вызываем обработчик этого таймера, исключаем структуру из списка активных и возвращаем в список свободных. Как и в листинге 3.15, странный вызов макроса `timercmp` обусловлен некорректной его реализацией в некоторых системах.

Вычисление времени следующего события

29-39 Если список активных таймеров не пуст, вычисляем разность между текущим моментом времени и временем срабатывания таймера в начале списка. Это значение передаем системному вызову `select`.

40-41 Если больше таймеров нет и нет ожидаемых событий ввода/вывода, то `tselect` возвращает управление. Обратите внимание, что возвращается нуль, тем самым извещается об отсутствии ожидающих событий. Семантика кода возврата отличается от семантики `select`.

42-43 Если нет событий таймера, но есть события ввода/вывода, то устанавливаем `tvp` в `NULL`, чтобы `select` не вернулся из-за тайм-аута.

Вызов select

44-48 Вызываем `select`, чтобы он дождался события. Если `select` завершается с ошибкой, то возвращаем код ошибки приложению. Если `select` возвращает положительное значение (произошло одно или более событий ввода/вывода), то возвращаем приложению число событий. Поскольку вызывали `select`, передавая указатели на маски событий, подготовленные приложением, то биты событий в них уже установлены.

49-54 Если select вернул нуль, то сработал один или несколько таймеров. Поскольку в этом случае select обнулит все маски событий, установленные приложением, восстановим их перед тем, как возвращаться к началу цикла, где вызываются обработчики таймеров.

Для вставки и удаления таймеров из списка был использован линейный поиск. При небольшом числе таймеров это не страшно, но при увеличении их числа производительность программы снижается, так как для поиска требуется O(n) операций, где n – число таймеров (для запуска обработчика события требуется время порядка O(1)). Вместо линейного поиска можно воспользоваться пирамидой [Sedgewick 1998] – для вставки, удаления и диспетчеризации требуется O(log n) операций – или хэширующим кольцом таймеров (hashing timing wheel) [Varghese and Lacuk 1997]; при этом эффективность может достигать O(1) для всех трех операций.

Заметим, что функция tselect не требует наличия ожидающих событий ввода/вывода, поэтому ее вполне можно использовать только как механизм организации тайм-аутов. В данном случае имеем следующие преимущества по сравнению с системным вызовом sleep:

- в системе UNIX sleep позволяет задерживать исполнение на интервал, кратный секунде, то есть разрешающая способность очень мала. В Windows такого ограничения нет. Во многих реализациях UNIX есть иные механизмы с более высокой степенью разрешения, однако они не очень распространены. Хотелось бы иметь механизм таймеров с высокой степенью разрешения, работающий на возможно большем числе платформ. Поэтому в UNIX принято использовать для реализации высокоточных таймеров вызов select;
- применение sleep или «чистого» select для организации нескольких таймеров затруднительно, поскольку требует введения дополнительных структур данных. В функции tselect все это уже сделано.

К сожалению, в Windows функция tselect в качестве таймера работает не совсем хорошо. В спецификации Winsock API [WinSock Group 1997] говорится, что использование select в качестве таймера «неудовлетворительно и не имеет оправданий». Хотя на это можно возразить, что «неудовлетворительность» – это когда системный вызов работает не так, как описано в опубликованной спецификации, все же придется придерживаться этой рекомендации. Тем не менее можно использовать функцию tselect и связанные с ней под Windows, только при этом следует указывать также и события ввода/вывода.

Резюме

В этом разделе обсуждены преимущества, которые дает управляемость приложения событиями. Разработан также обобщенный механизм работы с таймерами, не накладывающий ограничений на количество таймеров.

Совет 21. Подумайте, не сделать ли приложение событийно-управляемым (2)

Здесь будет продолжено обсуждение, начатое в совете 20, а также проиллюстрировано использование функции tselect в приложениях и рассмотрены некоторые

другие аспекты событийно-управляемого программирования. Вернемся к архитектуре с двумя соединениями из совета 19.

Взглянув на программу xout2 (листинг 3.13), вы увидите, что она не управляется событиями. Отправив сообщение удаленному хосту, вы не возвращаетесь к чтению новых данных из стандартного ввода, пока не придет подтверждение. Причина в том, что таймер может сбросить новое сообщение. Если бы вы взвели таймер для следующего сообщения, не дождавшись подтверждения, то никогда не узнали бы, подтверждено старое сообщение или нет.

Проблема, конечно, в том, что в программе xout2 только один таймер и поэтому она не может ждать более одного сообщения в каждый момент. Воспользовавшись tselect, вы сможете получить несколько таймеров из одного, предоставляемого select.

Представьте, что внешняя система из совета 19 – это шлюз, отправляющий сообщение третьей системе по ненадежному протоколу. Например, он мог бы посылать датаграммы в радиорелейную сеть. Предположим, что сам шлюз не дает информации о том, было ли сообщение успешно доставлено. Он просто переправляет сообщение и возвращает подтверждение, полученное от третьей системы.

Чтобы в какой-то мере обеспечить надежность, новый писатель xout3 повторно посылает сообщение (но только один раз), если в течение определенного времени не получает подтверждения. Если и второе сообщение не подтверждено, xout3 протоколирует этот факт и отбрасывает сообщение. Чтобы ассоциировать подтверждение с сообщением, на которое оно поступило, xout3 включает в каждое сообщение некий признак. Конечный получатель сообщения возвращает этот признак в составе подтверждения. Начнем с рассмотрения секции объявлений xout3 (листинг 3.18).

Листинг 3.18. Объявления для программы xout3

xout3.c

```
#define ACK     0x6    /* Символ подтверждения ACK. */
#define MRSZ    128    /* Максимальное число неподтвержденных */
                       /* сообщений. */
#define T1      3000   /* Ждать 3 с до первого ACK */
#define T2      5000   /* и 5 с до второго ACK. */
#define ACKSZ ( sizeof( u_int32_t ) + 1 )
typedef struct         /* Пакет данных. */
{
   u_int32_t len;      /* Длина признака и данных. */
   u_int32_t cookie;   /* Признак сообщения. */
   char buf[ 128 ];    /* Сообщение. */
} packet_t;

typedef struct         /* Структура сообщения. */
{
packet_t pkt;          /* Указатель на сохраненное сообщение. */
int id;                /* Идентификатор таймера. */
} msgrec_t;

static msgrec_t mr[ MRSZ ];
static SOCKET s;
```

xout3.c

Объявления

5 Признак, включаемый в каждое сообщение, – это 32-разрядный порядковый номер сообщения. Подтверждение от удаленного хоста определяется как ASCII-символ ACK, за которым следует признак подтверждаемого сообщения. Поэтому константа `ASCZ` вычисляется как длина признака плюс 1.

6-11 Тип `packet_t` определяет структуру посылаемого пакета. Поскольку сообщения могут быть переменной длины, в каждый пакет включена длина сообщения. Удаленное приложение может использовать это поле для разбиения потока данных на отдельные записи (об этом шла речь в совете 6). Поле `len` – это общая длина самого сообщения и признака. Проблемы, связанные с упаковкой структур, рассматриваются в замечаниях после листинга 2.15.

12-16 Структура `msgrec_t` содержит структуру `packet_t`, посланную удаленному хосту. Пакет сохраняется на случай, если придется послать его повторно. Поле `id` – это идентификатор таймера, выступающего в роли таймера ретрансмиссии для этого сообщения.

17 С каждым неподтвержденным сообщением связана структура `msgrec_t`. Все они хранятся в массиве `mr`.

Теперь обратимся к функции `main` программы `xout3` (листинг 3.19).

Листинг 3.19. Функция main программы xout3

xout3.c

```
int main( int argc, char **argv )
{
   fd_set allreads;
   fd_set readmask;
   msgrec_t *mp;
   int rc;
   int mid;
   int cnt = 0;
   u_int32_t msgid = 0;
   char ack[ ACKSZ ];

   INIT();
   s = tcp_client( argv[ 1 ], argv[ 2 ] );
   FD_ZERO( &allreads );
   FD_SET( s, &allreads );
   FD_SET( 0, &allreads );
   for ( mp = mr; mp < mr + MRSZ; mp++ )
      mp->pkt.len = -1;
   for ( ;; )
   {
      readmask = allreads;
      rc = tselect( s + 1, &readmask, NULL, NULL );
      if ( rc < 0 )
         error( 1, errno, "ошибка вызова tselect" );
```

```
24        if ( rc == 0 )
25           error( 1, 0, "tselect сказала, что нет событий\n" );
26        if ( FD_ISSET( s, &readmask ) )
27        {
28           rc = recv( s, ack + cnt, ACKSZ - cnt, 0 );
29           if ( rc == 0 )
30              error( 1, 0, "сервер отсоединился\n" );
31           else if ( rc < 0 )
32              error( 1, errno, "ошибка вызова recv" );
33           if ( ( cnt += rc ) < ACKSZ ) /* Целое сообщение? */
34              continue;    /* Нет, еще подождем. */
35           cnt = 0;     /* В следующий раз новое сообщение. */
36           if ( ack[ 0 ] != ACK )
37           {
38              error( 0, 0, "предупреждение: неверное подтверждение\n" )
39              continue;
40           }
41           memcpy( &mid, ack + 1, sizeof( u_int32_t ) );
42           mp = findmsgrec( mid );
43           if ( mp != NULL )
44           {
45           untimeout( mp->id );  /* Отменить таймер. */
46           freemsgrec( mp ); /* Удалить сохраненное сообщение. */
47           }
48        }
49        if ( FD_ISSET( 0, &readmask ) )
50        {
51           mp = getfreerec();
52           rc = read( 0, mp->pkt.buf, sizeof( mp->pkt.buf ) );
53           if ( rc < 0 )
54              error( 1, errno, "ошибка вызова read" );
55           mp->pkt.buf[ rc ] = '\0';
56           mp->pkt.cookie = msgid++;
57           mp->pkt.len = htonl( sizeof( u_int32_t ) + rc );
58           if ( send( s, &mp->pkt,
59                2 * sizeof( u_int32_t ) + rc, 0 ) < 0 )
60              error( 1, errno, "ошибка вызова send" );
61           mp->id = timeout( ( tofunc_t )lost_ACK, mp, T1 );
62        }
63     }
64 }
```

xout3.c

Инициализация

11-15 Так же, как и в программе xout2, соединяемся с удаленным хостом и инициализируем маски событий для tselect, устанавливая в них биты для дескрипторов stdin и сокета, который возвратила tcp_client.

16-17 Помечаем все структуры msgrec_t как свободные, записывая в поле длины пакета –1.

18-25 Вызываем `tselect` точно так же, как `select`, только не передаем последний параметр (времени ожидания). Если `tselect` возвращает ошибку или нуль, то выводим диагностическое сообщение и завершаем программу. В отличие от `select` возврат нуля из `tselect` – свидетельство ошибки, так как все тайм-ауты обрабатываются внутри.

Обработка входных данных из сокета

26-32 При получении события чтения из сокета ожидаем подтверждение. В совете 6 говорилось о том, что нельзя применить `recv` в считывании `ASCZ` байт, поскольку, возможно, пришли еще не все данные. Нельзя воспользоваться и функцией типа `readn`, которая не возвращает управления до получения указанного числа байт, так как это противоречило бы событийно-управляемой архитектуре приложения, – ни одно событие не может быть обработано, пока `readn` не вернет управления. Поэтому пытаемся прочесть столько данных, сколько необходимо для завершения обработки текущего подтверждения. В переменной `cnt` хранится число ранее прочитанных байт, поэтому `ASCZ - cnt` – это число недостающих байт.

33-35 Если общее число прочитанных байт меньше `ASCZ`, то возвращаемся к началу цикла и назначаем `tselect` ожидание прихода следующей партии данных или иного события. Если после только что сделанного вызова `recv` подтверждение получено, то сбрасываем `cnt` в нуль в ожидании следующего подтверждения (к этому моменту не было прочитано еще ни одного байта следующего подтверждения).

36-40 Далее, в соответствии с советом 11, выполняем проверку правильности полученных данных. Если сообщение – некорректное подтверждение, печатаем диагностическое сообщение и продолжаем работу. Возможно, здесь было бы правильнее завершить программу, так как удаленный хост послал неожиданные данные.

41-42 Наконец, извлекаем из подтверждения признак сообщения, вызываем `findmsgrec` для получения указателя на структуру `msgrec_t`, ассоциированную с сообщением, и используем ее для отмены таймера, после чего освобождаем `msgrec_t`. Функции `findmsgrec` и `freemsgrec` приведены в листинге 3.20.

Обработка данных из стандартного ввода

51-57 Когда `tselect` сообщает о событии ввода из `stdin`, получаем структуру `msgrec_t` и считываем сообщение в пакет данных. Присваиваем сообщению порядковый номер, пользуясь счетчиком `msgid`, и сохраняем его в поле `cookie` пакета. Обратите внимание, что вызывать `htonl` не нужно, так как удаленный хост не анализирует признак, а возвращает его без изменения. Записываем в поля пакета полную длину сообщения вместе с признаком. На этот раз вызываем `htonl`, так как удаленный хост использует это поле для чтения оставшейся части сообщения (совет 28).

58-61 Посылаем подготовленный пакет удаленному хосту и взводим таймер ретрансмиссии, обращаясь к функции `timeout`.

Оставшиеся функции программы `xout3` приведены в листинге 3.20.

Листинг 3.20. Вспомогательные функции программы xout3

xout3.c

```
msgrec_t *getfreerec( void )
{
   msgrec_t *mp;

   for ( mp = mr; mp < mr + MRSZ; mp++ )
      if ( mp->pkt.len == -1 ) /* Запись свободна? */
         return mp;
   error( 1, 0, "getfreerec: исчерпан пул записей сообщений\n" );
   return NULL; /* "Во избежание предупреждений компилятора. */
}

   msgrec_t *findmsgrec( u_int32_t mid )
{
   msgrec_t *mp;

   for ( mp = mr; mp < mr + MRSZ; mp++ )
      if ( mp->pkt.len != -1 && mp->pkt.cookie == mid )
         return mp;
   error( 0, 0,
   "findmsgrec: нет сообщения, соответствующего ACK %d\n", mid );
   return NULL;
}

void freemsgrec( msgrec_t *mp )
{
   if ( mp->pkt.len == -1 )
   error( 1, 0, "freemsgrec: запись сообщения уже освобождена\n" );
   mp->pkt.len = -1;
}

 static void drop( msgrec_t *mp )
{
   error( 0, 0, "Сообщение отбрасывается:   %s", mp->pkt.buf )
   freemsgrec( mp );
}

static void lost_ACK( msgrec_t *mp )
{
   error( 0, 0, "Повтор сообщения:   %s", mp->pkt.buf );
   if ( send( s, &mp->pkt,
        sizeof( u_int32_t ) + ntohl( mp->pkt.len ), 0 ) < 0 )
      error( 1, errno, "потерян ACK: ошибка вызова send" );
   mp->id = timeout( ( tofunc_t )drop, mp, T2 );
}
```

xout3.c

getfreerec

1-9 Данная функция ищет свободную запись в таблице `mr`. Просматриваем последовательно весь массив, пока не найдем пакет с длиной –1. Это означает, что запись свободна. Если бы массив `mr` был больше, то можно было бы завести список свободных, как было сделано для записей типа `tevent_t` в листинге 3.15.

findmsgrec

10-18 Эта функция почти идентичная `getfreerec`, только на этот раз ищем запись с заданным признаком сообщения.

freemsgrec

19-24 Убедившись, что данная запись занята, устанавливаем длину пакета в –1, помечая тем самым, что теперь она свободна.

drop

25-29 Данная функция вызывается, если не пришло подтверждение на второе посланное сообщение (см. `lost_ACK`). Пишем в протокол диагностику и отбрасываем запись, вызывая `freemsgrec`.

lost_ACK

30-37 Эта функция вызывается, если не пришло подтверждение на первое сообщение. Посылаем сообщение повторно и взводим новый таймер ретрансмиссии, указывая, что при его срабатывании надо вызвать функцию `drop`.

Для тестирования `xout3` напишем серверное приложение, которое случайным образом отбрасывает сообщения. Назовем этот сервер `extsys` (сокращение от external system – внешняя система). Его текст приведен в листинге 3.21.

Листинг 3.21. Внешняя система

extsys.c

```
#include "etcp.h"

#define COOKIESZ   4   /* Так установлено клиентом. */

int main( int argc, char **argv )
{
   SOCKET s;
   SOCKET s1;
   int rc;
   char buf[ 128 ];

   INIT();
   s = tcp_server( NULL, argv[ 1 ] );
   s1 = accept( s, NULL, NULL );
   if ( !isvalidsock( s1 ) )
      error( 1, errno, "ошибка вызова accept" );
   srand( 127 );
   for ( ;; )
```

```
16    {
17        rc = readvrec( s1, buf, sizeof( buf ) );
18        if ( rc == 0 )
19           error( 1, 0, "клиент отсоединился\n" );
20        if ( rc < 0 )
21           error( 1, errno, "ошибка вызова recv" );
22        if ( rand() % 100 < 33 )
23           continue;
24        write( 1, buf + COOKIESZ, rc - COOKIESZ );
25        memmove( buf + 1, buf, COOKIESZ );
26        buf[ 0 ] = '\006';
27        if ( send( s1, buf, 1 + COOKIESZ, 0 ) < 0 )
28           error( 1, errno, "ошибка вызова send" );
29    }
30 }
```

extsys.c

Инициализация

9-14 Выполняем обычную инициализацию сервера и вызываем функцию `srand` для инициализации генератора случайных чисел.

Примечание *Функция `rand` из стандартной библиотеки С работает быстро и проста в применении, но имеет ряд нежелательных свойств. Хотя для демонстрации `xout3` она вполне пригодна, но для серьезного моделирования нужно было бы воспользоваться более развитым генератором случайных чисел [Knuth 1998].*

17-21 Вызываем функцию `readvrec` для чтения записи переменной длины, посланной `xout3`.

22-23 Случайным образом отбрасываем примерно треть получаемых сообщений.

24-28 Если сообщение не отброшено, то выводим его на `stdout`, сдвигаем в буфере признак на один символ вправо, добавляем в начало символ АСK и возвращаем подтверждение клиенту.

Вы тестировали `xout3`, запустив `extsys` в одном окне и воспользовавшись конвейером из совета 20 в другом (рис. 3.7).

Можно сделать следующие замечания по поводу работы `xout3`:

- доставка сообщений по порядку не гарантирована. На примере сообщений 17 и 20 на рис. 3.8 вы видите, что повторно посланное сообщение нарушило порядок;
- можно было увеличить число повторных попыток, добавив счетчик попыток в структуру `msgrec_t` и заставив функцию `lost_ACK` продолжать попытки отправить сообщение до исчерпания счетчика;
- легко модифицировать `xout3` так, чтобы она работала по протоколу UDP, а не TCP. Это стало бы первым шагом на пути предоставления надежного UDP-сервиса (совет 8);
- если бы приложение работало с большим числом сокетов (и использовало функцию `tselect`), то имело бы смысл вынести встроенный код `readr`

в отдельную функцию. Такая функция могла бы получать на входе структуру, содержащую cnt, указатель на буфер ввода (или сам буфер) и адрес функции, которую нужно вызвать после получения полного сообщения;
- в качестве примера xout3, пожалуй, выглядит чересчур искусственно, особенно в контексте совета 19, но так или иначе она иллюстрирует, как можно решить задачу, часто возникающую на практике.

```
bsd $ mp | xout3 localhost 9000
xout3: Повтор сообщения:   message 3
xout3: Повтор сообщения:   message 4
xout3: Повтор сообщения:   message 5
xout3: Сообщение отбрасывается: message 4
xout3: Сообщение отбрасывается: message 5
xout3: Повтор сообщения:   message 11
xout3: Повтор сообщения:   message 14
xout3: Сообщение отбрасывается: message 11
xout3: Повтор сообщения:   message 16
xout3: Повтор сообщения:   message 17
xout3: Сообщение отбрасывается: message 14
xout3: Повтор сообщения:   message 19
xout3: Повтор сообщения:   message 20
xout3: Сообщение отбрасывается: message 16
xout3: Сервер отсоединился
Broken pipe
bsd $
```

```
bsd $ extsys 9000
message 1
message 2
message 3
message 6
message 7
message 8
message 9
message 10
message 12
message 13
message 15
message 18
message 17
message 21
message 20
message 23
^C сервер остановлен
bsd $
```

Рис. 3.7. Демонстрация xout 3

Резюме

В этом и предыдущем разделах говорилось о событийно-управляемом программировании и о том, как использовать вызов select для реагирования на события по мере их поступления. В совете 20 разработана функция tselect, позволившая получить несколько логических таймеров из одного физического. Эта функция и используемые с ней функции timeout и untimeout дают возможность задавать тайм-ауты сразу для нескольких событий, инкапсулируя внутри себя все сопутствующие этому детали.

Здесь была использована функция tselect, чтобы усовершенствовать пример из совета 19. Применение tselect позволило задавать отдельные таймеры ретрансмиссии для каждого сообщения, посланного ненадежному удаленному хосту через сервер-шлюз xout3.

Совет 22. Не прерывайте состояние TIME-WAIT для закрытия соединения

В этом разделе рассказывается о том, что такое состояние TIME-WAIT в протоколе TCP, для чего оно служит и почему не следует пытаться обойти его.

Поскольку состояние TIME-WAIT запрятано глубоко в недрах конечного автомата, управляющего работой TCP, многие программисты только подозревают о его существовании и смутно представляют себе назначение и важность этого состояния. Писать приложения TCP/IP можно, ничего не зная о состоянии TIME-WAIT, но необходимо разобраться в странном, на первый взгляд, поведении приложения (совет 23). Это позволит избежать непредвиденных последствий.

Рассмотрим состояние TIME-WAIT и определим, каково его место в работе TCP-соединения. Затем будет рассказано о назначении этого состояния и его важности, а также, почему и каким образом некоторые программисты пытаются обойти это состояние. В конце дано правильное решение этой задачи.

Что это такое

Состояние TIME-WAIT наступает в ходе разрыва соединения. Помните (совет 7), что для разрыва TCP-соединения нужно обычно обменяться четырьмя сегментами, как показано на рис. 3.8.

На рис. 3.8 показано соединение между двумя приложениями, работающими на хостах 1 и 2. Приложение на хосте 1 закрывает свою сторону соединения, при этом TCP посылает сегмент FIN хосту 2. Хост 2 подтверждает FIN сегментом ACK и доставляет FIN приложению в виде признака конца файла EOF (предполагается, что у приложения есть незавершенная операция чтения, – совет 16). Позже приложение на хосте 2 закрывает свою сторону соединения, посылая FIN хосту 1, который отвечает сегментом ACK.

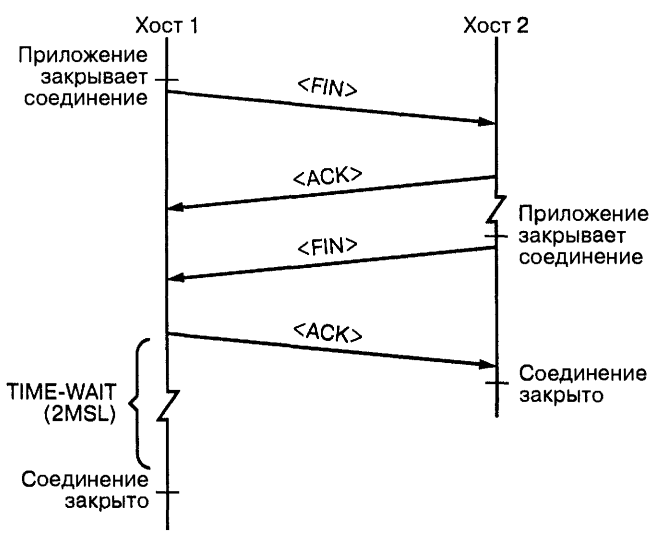


Рис. 3.8 Разрыв соединения

В этот момент хост 2 окончательно закрывает соединение и освобождает ресурсы. С точки зрения хоста 2, соединения больше не существует. Однако хост 1 не закрывает соединение, а переходит в состояние TIME-WAIT и остается в нем в течение двух максимальных продолжительностей существования сегмента (2MSL – maximum segment lifetime).

Примечание *Максимальное время существования сегмента (MSL) – это максимальное время, в течение которого сегмент может оставаться в сети, прежде чем будет уничтожен. В каждой IP-датаграмме есть поле TTL (time-to-live – время жизни). Это поле уменьшается на единицу каждым маршрутизатором, через который проходит датаграмма. Когда TTL становится равным нулю, датаграмма уничтожается. Хотя официально TTL измеряется в секундах, в действительности это поле почти всегда интерпретируется маршрутизаторами как счетчик промежуточных узлов. В RFC 1812 [Baker 1995] этот вопрос обсуждается подробнее.*

Прождав время 2MSL, хост 1 также закрывает соединение и освобождает ресурсы.

Относительно состояния TIME-WAIT следует помнить следующее:

- обычно в состояние TIME-WAIT переходит только одна сторона – та, что выполняет активное закрытие;

Примечание *Под активным закрытием понимается отправка первого FIN. Считается, что вторая сторона при этом выполняет пассивное закрытие. Возможно также одновременное закрытие, когда обе стороны закрывают соединение примерно в одно время, поэтому посланные ими FIN одновременно находятся в сети. В этом случае активное закрытие выполняют обе стороны, так что обе переходят в состояние TIME-WAIT.*

- в RFC 793 [Postel 1981b] MSL определено равным 2 мин. При этом соединение должно оставаться в состоянии TIME-WAIT в течение 4 мин. На практике это обычно не так. Например, в системах, производных от BSD, MSL равно 30 с, так что состояние TIME-WAIT длится всего 1 мин. Можно встретить и другие значения в диапазоне от 30 с до 2 мин;
- если в то время, когда соединение находится в состоянии TIME-WAIT, прибывает новый сегмент, то таймер на 2MSL перезапускается. Это будет рассматриваться ниже.

Зачем нужно состояние TIME-WAIT

Состояние TIME-WAIT служит двум целям:

- не дать соединению пропасть при потере последнего АСК, посланного активной стороной, в результате чего другая сторона повторно посылает FIN;
- дать время исчезнуть «заблудившимся сегментам», принадлежащим этому соединению.

Рассмотрим каждую из этих причин. В момент, когда сторона, выполняющая активное закрытие, готова подтвердить посланный другой стороной FIN, все данные, отправленные другой стороной, уже получены. Однако последний АСК может потеряться. Если это произойдет, то сторона, выполняющая пассивное закрытие,

обнаружит тайм-аут и пошлет свой FIN повторно (так как не получила ACK на последний порядковый номер).

А теперь посмотрим, что случится, если активная сторона не перейдет в состояние TIME-WAIT, а просто закроет соединение. Когда прибывает повторно переданный FIN, у TCP уже нет информации о соединении, поэтому он посылает в ответ RST (сброс), что для другой стороны служит признаком ошибки, а не нормального закрытия соединения. Но, так как сторона, пославшая последний ACK, все-таки перешла в состояние TIME-WAIT, информация о соединении еще хранится, так что она может корректно ответить на повторно отправленный FIN.

Этим объясняется и то, почему 2MSL-таймер перезапускается, если в состоянии TIME-WAIT приходит новый сегмент. Если последний ACK потерян, и другая сторона повторно послала FIN, то сторона, находящаяся в состоянии TIME-WAIT, еще раз подтвердит его и перезапустит таймер на случай, если и этот ACK будет потерян.

Второе назначение состояния TIME-WAIT более важно. Поскольку IP-датаграммы могут теряться или задерживаться в глобальной сети, TCP использует механизм подтверждений для своевременной повторной передачи неподтвержденных сегментов (совет 1). Если датаграмма просто задержалась в пути, но не потеряна, или потерян подтверждающий ее сегмент ACK, то после прибытия исходных данных могут поступить также и повторно переданные. TCP в этом случае определяет, что порядковые номера поступивших данных находятся вне текущего окна приема, и отбрасывает их.

А что случится, если задержавшийся или повторно переданный сегмент придет после закрытия соединения? Обычно это не проблема, так как TCP просто отбросит данные и пошлет RST. Когда RST дойдет до хоста, отправившего задержавшийся сегмент, то также будет отброшен, поскольку у этого хоста больше нет информации о соединении. Однако если между этими двумя хостами установлено новое соединение с такими же номерами портов, что и раньше, то заблудившийся сегмент может быть принят как принадлежащий новому соединению. Если порядковые номера данных в заблудившемся сегменте попадают в текущее окно приема нового соединения, то данные будут приняты, следовательно, новое соединение – скомпрометировано.

Состояние TIME-WAIT предотвращает такую ситуацию, гарантируя, что два прежних сокета (два IP-адреса и соответствующие им номера портов) повторно не используются, пока все сегменты, оставшиеся от старого соединения, не будут уничтожены. Таким образом, вы видите, что состояние TIME-WAIT играет важную роль в обеспечении надежности протокола TCP. Без него TCP не мог бы гарантировать доставку данных по порядку и без искажений (совет 9).

Принудительная отмена состояния TIME-WAIT

К сожалению, иногда можно досрочно выйти из состояния TIME-WAIT. Это называется *принудительной отменой* (TIME-WAIT assassination) и бывает случайно или намеренно.

Сначала посмотрим, как это может произойти случайно. По стандартам RFC 793, если соединение находится в состоянии TIME-WAIT и приходит RST, то соединение должно быть немедленно закрыто. Предположим, что имеется

соединение в состоянии TIME-WAIT и приходит старый сегмент-дубликат, который TCP не принимает (например, потому, что порядковый номер оказался вне окна приема). TCP посылает в ответ ACK, в котором указано, какой порядковый номер он ожидает (следующий за номером сегмента FIN, посланного другой стороной). Но у хоста на другой стороне уже нет информации о соединении, поэтому на этот ACK он отвечает сегментом RST. Когда этот RST приходит хосту, у которого соединение находится в состоянии TIME-WAIT, тот немедленно закрывает соединение, – состояние TIME-WAIT принудительно отменено.

Эта ситуация описана в RFC 1337 [Braden 1992b], где также рассматриваются трудности, сопряженные с принудительной отменой состояния TIME-WAIT. Опасность состоит в возможности «воскрешения» старого соединения (то есть появления соединения с теми же двумя сокетами), что может привести к подтверждению старых данных, десинхронизации соединения с входом в бесконечный цикл и к ошибочному завершению нового соединения.

Это легко предотвратить, изменив протокол TCP так, чтобы в состоянии TIME-WAIT было разрешено игнорировать RST. Хотя такое изменение, рекомендованное в RFC 1337, официально не одобрено, тем не менее в некоторых стеках оно реализовано.

Принудительно отменить состояние TIME-WAIT можно и намеренно. С помощью опции сокета SO_LINGER программист требует немедленного закрытия соединения даже в том случае, когда приложение выполняет активное закрытие. Этот сомнительный прием иногда рекомендуют применять, чтобы вывести «упавший» сервер из состояния TIME-WAIT и запустить его заново. Подробнее об этой проблеме и более правильном способе ее решения будет рассказано в совете 23. Корректно написанное приложение *никогда* не должно манипулировать состоянием TIME-WAIT, поскольку это неотъемлемая часть механизма обеспечения надежности TCP.

Обычно, когда приложение закрывает соединение, вызов close или closesocket возвращается немедленно, даже если в буфере передачи еще есть данные. Разумеется, TCP будет пытаться доставить эти данные, но приложение не имеет информации, удалось ли это. Чтобы решить эту проблему, можно установить опцию сокета SO_LINGER. Для этого следует заполнить структуру linger и вызывать setsockopt с параметром SO_LINGER.

В большинстве UNIX-систем структура linger определена в заголовочном файле /usr/include/sys/socket.h. В системе Windows она находится в файле winsock2.h. В любом случае она выглядит так:

```
struct linger {
   int l_onoff;    /* Включить/выключить опцию. */
   int l_linger;   /* Время задержки. */
};
```

Если поле l_onoff равно нулю, то опция задержки отключается, и выбирается поведение по умолчанию – вызов close или closesocket возвращается немедленно, а ядро продолжает попытки доставить еще не переданные данные. Если же l_onoff не равно нулю, то работа зависит от значения поля l_linger. Если

l_linger отлично от нуля, то считается, что это время, в течение которого ядро должно подождать отправки и подтверждения оставшихся в буфере передачи данных. При этом close или closesocket не возвращается, пока данные не будут доставлены или не истечет указанное время.

Если к моменту завершения ожидания данные еще не доставлены, то close или closesocket возвращает код EWOULDBLOCK, и недоставленные данные могут быть потеряны. Если все данные уже доставлены, то оба вызова возвращают нуль.

Примечание *К сожалению, семантика поля l_linger зависит от реализации. В Windows и некоторых реализациях UNIX это число секунд, на которое следует задержать закрытие сокета. В системах, производных от BSD, это число тактов таймера (хотя в документации сказано, что это число секунд).*

Используя опцию SO_LINGER таким способом, вы гарантируете, что данные будут доставлены уровню TCP на удаленном хосте. Но они не обязательно будут прочитаны приложением. Более правильный способ добиться последнего – использовать процедуру аккуратного размыкания, описанную в совете 16.

Если поле l_linger равно нулю, то соединение разрывается. Это означает, что хосту на другом конце посылается RST, и соединение закрывается немедленно, не переходя в состояние TIME-WAIT. Это преднамеренная принудительная отмена состояния TIME-WAIT, о которой упоминалось выше. Как было сказано, это опасный прием, который в обычном приложении применять не следует.

Резюме

В этом разделе обсуждено состояние TIME-WAIT, которое часто понимают неправильно. Это состояние – важная часть механизма обеспечения надежности протокола TCP, и попытки обойти его неверны. Преждевременный выход из состояния TIME-WAIT может быть обусловлен «естественным» стечением обстоятельств в сети или программой, которая манипулирует опцией SO_LINGER.

Совет 23. Сервер должен устанавливать опцию SO_REUSEADDR

В сетевых конференциях очень часто задают вопрос: «Когда сервер «падает» или нормально завершает сеанс, я пытаюсь его перезапустить и получаю ошибку «Address already in use». А через несколько минут сервер перезапускается нормально. Как сделать так, чтобы сервер рестартовал немедленно?» Чтобы проиллюстрировать эту проблему, напишем сервер эхо-контроля, который будет работать именно так (листинг 3.22).

Листинг 3.22. Некорректный сервер эхо-контроля

badserver.c

```
#include "etcp.h"
int main( int argc, char **argv )
```

```
{
   struct sockaddr_in local;
   SOCKET s;
   SOCKET s1;
   int rc;
   char buf[ 1024 ];

   INIT();
   s = socket( PF_INET, SOCK_STREAM, 0 );
   if ( !isvalidsock( s ) )
      error( 1, errno, "Не могу получить сокет" );
   bzero( &local, sizeof( local ) );
   local.sin_family = AF_INET;
   local.sin_port = htons( 9000 );
   local.sin_addr.s_addr = htonl( INADDR_ANY );
   if ( bind( s, ( struct sockaddr * )&local,
      sizeof( local ) ) < 0 )
      error( 1, errno, "Не могу привязать сокет" );
   if ( listen( s, NLISTEN ) < 0 )
      error( 1, errno, "ошибка вызова listen" );
   s1 = accept( s, NULL, NULL );
   if ( !isvalidsock( s1 ) )
      error( 1, errno, "ошибка вызова accept" );
   for ( ;; )
   {
      rc = recv( s1, buf, sizeof( buf ), 0 );
      if ( rc < 0 )
         error( 1, errno, "ошибка вызова recv" );
      if ( rc == 0 )
         error( 1, 0, "Клиент отсоединился\n" );
      rc = send( s1, buf, rc, 0 );
      if ( rc < 0 )
         error( 1, errno, "ошибка вызова send" );
   }
}
```

badserver.c

На первый взгляд, сервер выглядит вполне нормально, только номер порта «зашит» в код. Если запустить его в одном окне и соединиться с ним с помощью программы telnet, запущенной в другом окне, то получится ожидаемый результат. (На рис. 3.9 опущены сообщения telnet об установлении соединения.)

Проверив, что сервер работает, останавливаете клиента, переходя в режим команд telnet и вводя команду завершения. Обратите внимание, что если немедленно повторить весь эксперимент, то будет тот же результат. Таким образом, `badserver` перезапускается без проблем.

А теперь проделайете все еще раз, но только остановите сервер. При попытке перезапустить сервер вы получите сообщение «Address already in use» (сообщение разбито на две строчки). Разница в том, что во втором эксперименте вы остановили сервер, а не клиент – рис. 3.10.

```
bsd $ badserver                         | bsd $ telnet localhost 9000
badserver: Клиент отсоединился          | hello
bsd $: badserver                        | hello
badserver: Клиент отсоединился          | ^]
bsd $                                   | telnet> quit Клиент завершил сеанс.
                                        | Connection closed
                                        |    Сервер перезапущен.
                                        | bsd $ telnet localhost 9000
                                        | world
                                        | world
                                        | ^]
                                        | telnet> quit Клиент завершил сеанс.
                                        | Connection closed
                                        | bsd $
```

Рис. 3.9. Завершение работы клиента

```
bsd $ badserver                         | bsd $ telnet localhost 9000
^C Сервер остановлен                    | hello again
bsd $ badserver                         | hello again
badserver: Не могу привязать сокет:     | Connection closed by
                                        | foreign host
Address already in use (48)             | bsd $
bsd $
```

Рис. 3.10. Завершение работы сервера

Чтобы разобраться, что происходит, нужно помнить о двух вещах:

- состоянии TIME-WAIT протокола TCP;
- TCP-соединение полностью определено четырьмя факторами (локальный адрес, локальный порт, удаленный адрес, удаленный порт).

Как было сказано в совете 22, сторона соединения, которая выполняет активное закрытие (посылает первый FIN), переходит в состояние TIME-WAIT и остается в нем в течение 2MSL. Это первый ключ к пониманию того, что вы наблюдали в двух предыдущих примерах: если активное закрытие выполняет клиент, то можно перезапустить обе стороны соединения. Если же активное закрытие выполняет сервер, то его рестартовать нельзя. TCP не позволяет это сделать, так как предыдущее соединение все еще находится в состоянии TIME-WAIT.

Если бы сервер перезапустился и с ним соединился клиент, то возникло бы новое соединение, возможно, даже с другим удаленным хостом. Как было сказано, TCP-соединение полностью определяется локальными и удаленными адресами и номерами портов, так что даже если с вами соединился клиент с того же удаленного хоста, проблемы не возникнет при другом номере удаленного порта.

Примечание *Даже если клиент с того же удаленного хоста воспользуется тем же номером порта, проблемы может и не возникнуть. Традиционно реализация BSD разрешает такое соединение, если*

только порядковый номер посланного клиентом сегмента SYN больше последнего порядкового номера, зарегистрированного соединением, которое находится в состоянии TIME-WAIT.

Возникает вопрос: почему TCP возвращает ошибку, когда делается попытка перезапустить сервер? Причина не в TCP, который требует только уникальности указанных факторов, а в API сокетов, требующем двух вызовов для полного определения этой четверки. В момент вызова bind еще неизвестно, последует ли за ним connect, и, если последует, то будет ли в нем указано новое соединение, или он попытается повторно использовать существующее. В книге [Torek 1994] автор – и не он один – предлагает заменить вызовы bind, connect и listen одной функцией, реализующей функциональность всех трех. Это даст возможность TCP выявить, действительно ли задается уже используемая четверка, не отвергая попыток перезапустить закрывшийся сервер, который оставил соединение в состоянии TIME-WAIT. К сожалению, элегантное решение Торека не было одобрено.

Но существует простое решение этой проблемы. Можно разрешить TCP привязку к уже используемому порту, задав опцию сокета SO_REUSEADDR. Чтобы проверить, как это работает, вставим между строками 7 и 8 файла badserver.c строку

```
const int on = 1;
```

а между строками 12 и 13 – строки

```
if ( setsockopt( s, SOL_SOCKET, SO_REUSEADDR, &on,
   sizeof( on ) ) )
   error( 1, errno, "ошибка вызова setsockopt" );
```

Заметьте, что вызов setsockopt должен *предшествовать* вызову bind. Если назвать исправленную программу goodserver и повторить эксперимент (рис. 3.11), то получите такой результат:

```
bsd $ goodserver              | bsd $ telnet localhost 9000
^C Сервер остановлен.         | hello once again
bsd $                         | hello once again
                              | Connection closed by foreign host
                              |    Сервер перезапущен.
                              | bsd $ telnet localhost 9000
                              | hello one last time
                              | hello one last time
```

Рис. 3.11. Завершение работы сервера, в котором используется опция SO_REUSEADDR

Теперь вы смогли перезапустить сервер, не дожидаясь выхода предыдущего соединения из состояния TIME-WAIT. Поэтому в сервере *всегда* надо устанавливать опцию сокета SO_REUSEADDR. Обратите внимание, что в предлагаемом каркасе и в функции tcp_server это уже делается.

Некоторые, в том числе авторы книг, считают, что задание опции SO_REUSEADDR опасно, так как позволяет TCP создать четверку, идентичную уже используемой, и таким образом создать проблему. Это ошибка. Например, если попытаться создать

два идентичных прослушивающих сокета, то TCP отвергнет операцию привязки, даже если вы зададите опцию SO_REUSEADDR:

```
bsd $ goodserver &
[1] 1883
bsd $ goodserver
goodserver: Не могу привязать сокет: Address already in use (48)
bsd $
```

Аналогично если вы привяжете одни и те же локальный адрес и порт к двум разным клиентам, задав SO_REUSEADDR, то bind для второго клиента завершится успешно. Однако на попытку второго клиента связаться с тем же удаленным хостом и портом, что и первый, TCP ответит отказом.

Помните, что нет причин, мешающих установке опции SO_REUSEADDR в сервере. Это позволяет перезапустить сервер сразу после его завершения. Если же этого не сделать, то сервер, выполнявший активное закрытие соединения, не перезапустится.

Примечание *В книге [Stevens 1998] отмечено, что с опцией SO_REUSEADDR связана небольшая проблема безопасности. Если сервер привязывает универсальный адрес INADDR_ANY, как это обычно и делается, то другой сервер может установить опцию SO_REUSEADDR и привязать тот же порт, но с конкретным адресом, «похитив» тем самым соединение у первого сервера. Эта проблема действительно существует, особенно для сетевой файловой системы (NFS) даже в среде UNIX, поскольку NFS привязывает порт 2049 из открытого всем диапазона. Однако такая опасность существует не из-за использования NFS опции SO_REUSEADDR, а потому что это может сделать другой сервер. Иными словами, эта опасность имеет место независимо от установки SO_REUSEADDR, так что это не причина для отказа от этой опции.*

Следует отметить, что у опции SO_REUSEADDR есть и другие применения. Предположим, например, что сервер работает на машине с несколькими сетевыми интерфейсами и ему необходимо иметь информацию, какой интерфейс клиент указал в качестве адреса назначения. При работе с протоколом TCP это легко, так как серверу достаточно вызвать getsockname после установления соединения. Но, если реализация TCP/IP не поддерживает опции сокета IP_RECVDSTADDR, то UDP-сервер так поступить не может. Однако UDP-сервер может решить эту задачу, установив опцию SO_REUSEADDR и привязав свой хорошо известный порт к конкретным, интересующим его интерфейсам, а универсальный адрес INADDR_ANY – ко всем остальным интерфейсам. Тогда сервер определит указанный клиентом адрес по сокету, в который поступила датаграмма.

Аналогичная схема иногда используется TCP- и UDP-серверами, которые хотят предоставлять разные варианты сервиса в зависимости от адреса, указанного клиентом. Допустим, вы хотите использовать свою версию tcpmux (совет 18) для предоставления одного набора сервисов, когда клиент соединяется с интерфейсом

по адресу 198.200.200.1, и другого – при соединении клиента с иным интерфейсом. Для этого запускаете экземпляр tcpmux со специальными сервисами на интерфейсе 198.200.200.1, а экземпляр со стандартными сервисами – на всех остальных интерфейсах, указав универсальный адрес INADDR_ANY. Поскольку tcpmux устанавливает опцию SO_REUSEADDR, TCP позволяет повторно привязать порт 1, хотя при второй привязке указан универсальный адрес.

И, наконец, SO_REUSEADDR используется в системах с поддержкой группового вещания, чтобы дать возможность одновременно нескольким приложениям прослушивать входящие датаграммы, вещаемые на группу. Подробнее это рассматривается в книге [Stevens 1998].

Резюме

В этом разделе рассмотрена опция сокета SO_REUSEADDR. Ее установка позволяет перезапустить сервер, от предыдущего «воплощения» которого еще осталось соединение в состоянии TIME-WAIT. Серверы должны всегда устанавливать эту опцию, которая не влечет угрозу безопасности.

Совет 24. По возможности пишите один большой блок вместо нескольких маленьких

Для этой рекомендации есть несколько причин. Первая очевидна и уже обсуждалась выше: каждое обращение к функциям записи (write, send и т.д.) требует, по меньшей мере, двух контекстных переключений, а это довольно дорогая операция. С другой стороны, многократные операции записи (если не считать случаев типа записи по одному байту) не требуют заметных накладных расходов в приложении. Таким образом, совет избегать лишних системных вызовов – это, скорее, «правила хорошего тона», а не острая необходимость.

Есть, однако, и более серьезная причина избегать многократной записи мелких блоков – эффект алгоритма Нейгла. Этот алгоритм кратко рассмотрен в совете 15. Теперь изучим его взаимодействие с приложениями более детально. Если не принимать в расчет алгоритм Нейгла, то это может привести к неправильному решению задач, так или иначе связанных с ним, и существенному снижению производительности некоторых приложений.

К сожалению, алгоритм Нейгла, равно как и состояние TIME-WAIT, многие программисты понимают недостаточно хорошо. Далее будут рассмотрены причины появления этого алгоритма, способы его отключения и предложены эффективные решения, которые обеспечивают хорошую производительность приложению, не оказывая негативного влияния на сеть в целом.

Алгоритм был впервые предложен в 1984 году Джоном Нейглом (RFC 896 [Nagle 1984]) для решения проблем производительности таких программ, как telnet и ей подобных. Обычно эти программы посылают каждое нажатие клавиши в отдельном сегменте, что приводит к засорению сети множеством крохотных датаграмм (tinygrams). Если принять во внимание, что минимальный размер TCP-сегмента (без данных) равен 40 байт, то накладные расходы при посылке одного байта в сегменте достигают 4000%. Но важнее то, что увеличивается число

пакетов в сети. А это приводит к перегрузке и необходимости повторной передачи, из-за чего перегрузка еще более увеличивается. В неблагоприятном случае в сети находится несколько копий каждого сегмента, и пропускная способность резко снижается по сравнению с номинальной.

Соединение считается простаивающим, если в нем нет неподтвержденных данных (то есть хост на другом конце подтвердил все отправленные ему данные). В первоначальном виде алгоритм Нейгла должен был предотвращать описанные выше проблемы. При этом новые данные от приложения не посылаются до тех пор, пока соединение не перейдет в состояние простоя. В результате в соединении не может находиться более одного небольшого неподтвержденного сегмента.

Процедура, описанная в RFC 1122 [Braden 1989] несколько ослабляет это требование, разрешая посылать данные, если их хватает для заполнения целого сегмента. Иными словами, если можно послать не менее MSS байт, то это разрешено, даже если соединение не простаивает. Заметьте, что условие Нейгла при этом попрежнему выполняется: в соединении находится не более одного небольшого неподтвержденного сегмента.

Многие реализации не следуют этому правилу буквально, применяя алгоритм Нейгла не к сегментам, а к операциям записи. Чтобы понять, в чем разница, предположим, что MSS составляет 1460 байт, приложение записывает 1600 байт, в окнах приема и передачи свободно, по меньшей мере, 2000 байт и соединение простаивает. Если применить алгоритм Нейгла к сегментам, то следует послать 1460 байт, а затем ждать подтверждения перед отправкой следующих 140 байт – алгоритм Нейгла применяется при посылке каждого сегмента. Если же использовать алгоритм Нейгла к операциям записи, то следует послать 1460 байт, а вслед за ними еще 140 байт – алгоритм применяется только тогда, когда приложение передает TCP новые данные для доставки.

Алгоритм Нейгла работает хорошо и не дает приложениям забить сеть крохотными пакетами. В большинстве случае производительность не хуже, чем в реализации TCP, в которой алгоритм Нейгла отсутствует.

Примечание *Представьте, например, приложение, которое передает TCP один байт каждые 200 мс. Если период кругового обращения (RTT) для соединения равен одной секунде, то TCP без алгоритма Нейгла будет посылать пять сегментов в секунду с накладными расходами 4000%. При наличии этого алгоритма первый байт отсылается сразу, а следующие четыре байта, поступившие от приложения, будут задержаны, пока не придет подтверждение на первый сегмент. Тогда все четыре байта посылаются сразу. Таким образом, вместо пяти сегментов послано только два, за счет чего накладные расходы уменьшились до 1600% при сохранении той же скорости 5 байт/с.*

К сожалению, алгоритм Нейгла может плохо взаимодействовать с другой, добавленной позднее возможностью TCP – отложенным подтверждением.

Когда прибывает сегмент от удаленного хоста, TCP задерживает отправку ACK в надежде, что приложение скоро ответит на только что полученные данные. Поэтому ACK можно будет объединить с данными. Традиционно в системах, производных от BSD, величина задержки составляет 200 мс.

Примечание *В RFC 1122 не говорится о сроке задержки, требуется лишь, чтобы она была не больше 500 мс. Рекомендуется также подтверждать, по крайней мере, каждый второй сегмент.*

Отложенное подтверждение служит той же цели, что и алгоритм Нейгла – уменьшить число повторно передаваемых сегментов.

Принцип совместной работы этих механизмов рассмотрим на примере типичного сеанса «запрос/ответ». Как показано на рис. 3.12, клиент посылает короткий запрос серверу, ждет ответа и посылает следующий запрос.

Заметьте, что алгоритм Нейгла не применяется, поскольку клиент не посылает новый сегмент, не дождавшись ответа на предыдущий запрос, вместе с которым приходит и ACK. На стороне сервера задержка подтверждения дает серверу время ответить. Поэтому для каждой пары запрос/ответ нужно всего два сегмента. Если через RTT обозначить период кругового обращения сегмента, а через T_p – время, необходимое серверу для обработки запроса и отправки ответа (в миллисекундах), то на каждую пару запрос/ответ уйдет RTT + T_p мс.

А теперь предположим, что клиент посылает свой запрос в виде двух последовательных операций записи. Часто причина в том, что запрос состоит из заголовка, за которым следуют данные. Например, клиент, который посылает серверу запросы переменной длины, может сначала послать длину запроса, а потом сам запрос.

Примечание *Пример такого типа изображен на рис. 2.17, но там были приняты меры для отправки длины и данных в составе одного сегмента.*

На рис. 3.13 показан поток данных.

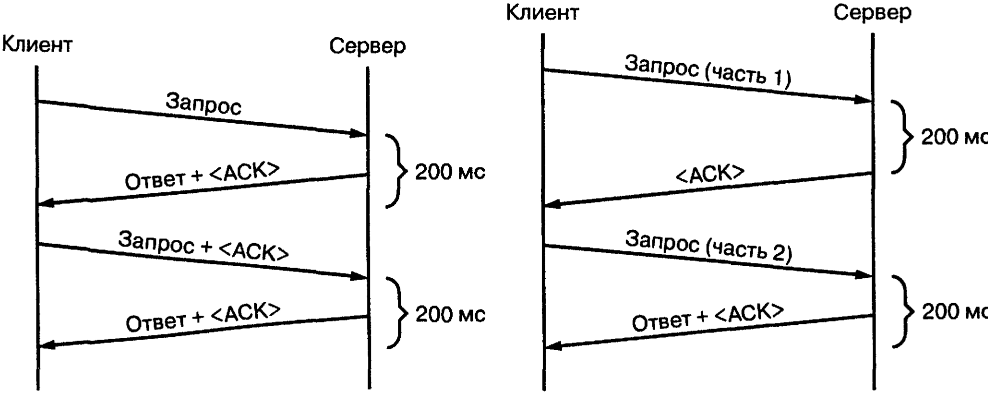


Рис. 3.12. Поток данных из одиночных сегментов сеанса «запрос/ответ»

Рис. 3.13. Взаимодействие алгоритма Нейгла и отложенного подтверждения

На этот раз алгоритмы взаимодействуют так, что число сегментов, посланных на каждую пару запрос/ответ, удваивается, и это вносит заметную задержку.

Данные из первой части запроса посылаются немедленно, но алгоритм Нейгла не дает послать вторую часть. Когда серверное приложение получает первую часть запроса, оно не может ответить, так как запрос целиком еще не пришел. Это значит, что перед посылкой подтверждения на первую часть должен истечь тайм-аут установленный таймером отложенного подтверждения. Таким образом, алгоритмы Нейгла и отложенного подтверждения блокируют друг друга: алгоритм Нейгла мешает отправке второй части запроса, пока не придет подтверждение на первую, а алгоритм отложенного подтверждения не дает послать АСК, пока не сработает таймер, поскольку сервер ждет вторую часть. Теперь для каждой пары запрос/ответ нужно четыре сегмента и $2 \times RTT + T_p + 200$ мс. В результате за секунду можно обработать не более пяти пар запрос/ответ, даже если забыть о времени обработки запроса сервером и о периоде кругового обращения.

Примечание *Для многих систем это изложение чрезмерно упрощенное. Например, системы, производные от BSD, каждые 200 мс проверяют все соединения, для которых подтверждение было отложено. При этом АСК посылается независимо от того, сколько времени прошло в действительности. Это означает, что реальная задержка может составлять от 0 до 200 мс, в среднем 100 мс. Однако часто задержка достигает 200 мс из-за «фазового эффекта», состоящего в том, что ожидание прерывается следующим тактом таймера через 200 мс. Первый же ответ синхронизирует ответы с тактовым генератором. Хороший пример такого поведения см. в работе [Minshall et al. 1999].*

Последний пример показывает причину проблемы: клиент выполняет последовательность операций «запись, запись, чтение». Любая такая последовательность приводит к нежелательной интерференции между алгоритмом Нейгла и алгоритмом отложенного подтверждения, поэтому ее следует избегать. Иными словами, приложение, записывающее небольшие блоки, будет страдать всякий раз, когда хост на другом конце сразу не отвечает.

Представьте приложение, которое занимается сбором данных и каждые 50 мс посылает серверу одно целое число. Если сервер не отвечает на эти сообщения, а просто записывает данные в журнал для последующего анализа, то будет наблюдаться такая же интерференция. Клиент, пославший одно целое, блокируется алгоритмом Нейгла, а затем алгоритмом отложенного подтверждения, например на 200 мс, после чего посылает по четыре целых каждые 200 мс.

Отключение алгоритма Нейгла

Поскольку сервер из последнего примера записывал в журнал полученные данные, взаимодействие алгоритмов Нейгла и отложенного подтверждение не принесло вреда и, по сути, уменьшило общее число пакетов в четыре раза. Допустим, что клиент посылает серверу серию результатов измерения температуры, и сервер

должен отреагировать в течение 100 мс, если показания выходят за пределы допустимого диапазона. В этом случае задержка на 200 мс, вызванная интерференцией алгоритмов, уже нетерпима, и ее надо устранить.

Хорошо, что RFC 1122 требует наличия метода, отключающего алгоритм Нейгла. Пример с клиентом, следящим за температурой, – это один из случаев, когда такое отключение необходимо. Менее драматичный, но более реалистичный пример относится к системе X-Window, работающей на платформе UNIX. Поскольку X использует протокол TCP для общения между дисплеем (сервером) и приложением (клиентом), X-сервер должен доставлять информацию о действиях пользователя (перемещении курсора мыши) X-клиенту без задержек, вносимых алгоритмом Нейгла.

В API сокетов можно отключить алгоритм Нейгла с помощью установки опции сокета TCP_NODELAY.

```
const int on = 1;
setsockopt( s, IPPROTO_TCP, TCP_NODELAY, &on, sizeof( on ) );
```

Но возможность отключения алгоритма Нейгла вовсе не означает, что это *обязательно* делать. Приложений, имеющих реальные основания отключать алгоритм, значительно меньше, чем тех, которые это делают без причины. Причина, по которой программисты с пугающей регулярностью сталкиваются с классической проблемой интерференции между алгоритмами Нейгла и отложенного подтверждения, в том, что делают много мелких операций записи вместо одной большой. Потом они замечают, что производительность приложений намного хуже, чем ожидалось, и спрашивают, что делать. И кто-нибудь обязательно ответит: «Это все алгоритм Нейгла. Отключите его!». Нет нужды говорить, что после отключения этого алгоритма проблема производительности действительно исчезает. Только за это приходится расплачиваться увеличением числа крохотных пакетов в сети. Если так работают многие приложения или, что еще хуже, алгоритм Нейгла отключен по умолчанию, то возрастет нагрузка сети. В худшем случае это может привести к полному затору.

Запись со сбором

Как видите, существуют приложения, которые, действительно, должны отключать алгоритм Нейгла, но в основном это делается из-за проблем с производительностью, причина которых в отправке логически связанных данных серией из отдельных операций записи. Есть много способов собрать данные, чтобы послать их вместе. Наконец всегда можно скопировать различные порции данных в один буфер, которые потом и передать операции записи. Но, как объясняется в совете 26, к такому методу следует прибегать в крайнем случае. Иногда можно организовать хранение данных в одном месте, как и сделано в листинге 2.15. Чаще, однако, данные находятся в нескольких несмежных буферах, а хотелось бы послать их одной операцией записи.

Для этого и в UNIX, и в Winsock предусмотрен некоторый способ. К сожалению, эти способы немного отличаются. В UNIX есть системный вызов writev и парный ему вызов readv. При использовании writev вы задаете список буферов,

из которых должны собираться данные. Это решает исходную задачу: можно размещать данные в нескольких буферах, а записывать их одной операцией, исключив тем самым интерференцию между алгоритмами Нейгла и отложенного подтверждения.

```
#include <sys/uio.h>

ssize_t writev( int fd, const struct iovec *iov, int cnt );

ssize_t readv( int fd, const struct iovec *iov, int cnt );
```

Возвращаемое значение: число переданных байт или –1 в случае ошибки.

Параметр *iov* – это указатель на массив структур `iovec`, в которых хранятся указатели на буферы данных и размеры этих буферов:

```
struct iovec {
   char *iov_base; /* Адрес начала буфера. */
   size_t iov_len; /* Длина буфера. */
};
```

Примечание *Это определение взято из системы FreeBSD. Теперь во многих системах адрес начала буфера определяется так:*

```
void *iov_base; /* адрес начала буфера */
```

Третий параметр, *cnt* – это число структур `iovec` в массиве (иными словами, количество буферов).

У вызовов `writev` и `readv` практически общий интерфейс. Их можно использовать для любых файловых дескрипторов, а не только для сокетов.

Чтобы это понять, следует переписать клиент (листинг 3.23), работающий с записями переменной длины (листинг 2.15), с использованием `writev`.

Листинг 3.23. Клиент, посылающий записи переменной длины с помощью writev

vrcv.c

```
#include "etcp.h"
#include <sys/uio.h>

int main( int argc, char **argv )
{
   SOCKET s;
   int n;
   char buf[ 128 ];
   struct iovec iov[ 2 ];

   INIT();
   s = tcp_client( argv[ 1 ], argv[ 2 ] );
   iov[ 0 ].iov_base = ( char * )&n;
   iov[ 0 ].iov_len = sizeof( n );
   iov[ 1 ].iov_base = buf;

   while ( fgets( buf, sizeof( buf ), stdin ) != NULL )
```

```
15    {
16        iov[ 1 ].iov_len = strlen( buf );
17        n = htonl( iov[ 1 ].iov_len );
18        if ( writev( s, iov, 2 ) < 0 )
19            error( 1, errno, "ошибка вызова writev" );
20    }
21    EXIT( 0 );
22 }
```

vrcv.c

Инициализация

9-13 Выполнив обычную инициализацию клиента, формируем массив iov. Поскольку в прототипе writev имеется спецификатор const для структур, на которые указывает параметр iov, то есть гарантия, что массив iov не будет изменен внутри writev, так что большую часть параметров можно задавать вне цикла while.

Цикл обработки событий

14-20 Вызываем fgets для чтения одной строки из стандартного ввода, вычисляем ее длину и записываем в поле структуры из массива iov. Кроме того, длина преобразуется в сетевой порядок байт и сохраняется в переменной n.

Если запустить сервер vrs (совет 6) и вместе с ним клиента vrcv, то получатся те же результаты, что и раньше.

В спецификации Winsock определен другой, хотя и похожий интерфейс.

```
#include <winsock2.h>

int WSAAPI WSAsend( SOCKET s, LPWSABUF, DWORD cnt,
    LPDWORD sent, DWORD flags, LPWSAOVERLAPPED ovl,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE func );
```

Возвращаемое значение: 0 в случае успеха, в противном случае SOCKET_ERROR.

Последние два аргумента используются при вводе/выводе с перекрытием, и в данном случае не имеют значения, так что обоим присваивается значение NULL. Параметр *buf* указывает на массив структур типа WSABUF, играющих ту же роль, что структуры iovec в вызове writev.

```
typedef struct _WSABUF {
    u_long len;          /* Длина буфера. */
    char FAR *   buf;    /* Указатель на начало буфера. */
} WSABUF, FAR * LPWSABUF;
```

Параметр *sent* – это указатель на переменную типа DWORD, в которой хранится число переданных байт при успешном завершении вызова. Параметр *flags* аналогичен одноименному параметру в вызове send.

Версия клиента, посылающего сообщения переменной длины, на платформе Windows выглядит так (листинг 3.24):

Листинг 3.24. Версия vrcv для Winsock

vrcvw.c

```
#include "etcp.h"

int main( int argc, char **argv )
{
    SOCKET s;
    int n;
    char buf[ 128 ];
    WSABUF wbuf[ 2 ];
    DWORD sent;

    INIT();
    s = tcp_client( argv[ 1 ], argv[ 2 ] );
    wbuf[ 0 ].buf = ( char * )&n;
    wbuf[ 0 ].len = sizeof( n );
    wbuf[ 1 ].buf = buf;
    while ( fgets( buf, sizeof( buf ), stdin ) != NULL )
    {
        wbuf[ 1 ].len = strlen( buf );
        n = htonl( wbuf[ 1 ].len );
        if ( WSASend( s, wbuf, 2, &sent, 0, NULL, NULL ) < 0 )
            error( 1, errno, "ошибка вызова WSASend" );
    }
    EXIT( 0 );
}
```

vrcvw.c

Как видите, если не считать иного обращения к вызову записи со сбором, то Winsock-версия идентична UNIX-версии.

Резюме

В этом разделе разобран алгоритм Нейгла и его взаимодействие с алгоритмом отложенного подтверждения. Приложения, записывающие в сеть несколько маленьких блоков вместо одного большого, могут заметно снизить производительность.

Поскольку алгоритм Нейгла помогает предотвратить действительно серьезную проблему – переполнение сети крохотными пакетами, не следует отключать его для повышения производительности приложений, выполняющих запись мелкими блоками. Вместо этого следует переписать приложение так, чтобы все логические связанные данные выводились сразу. Здесь был рассмотрен удобный способ решения этой задачи с помощью системного вызова `writev` в UNIX или `WSASend` в Winsock.

Совет 25. Научитесь организовывать тайм-аут для вызова connect

В совете 7 отмечалось, что для установления TCP-соединения стороны обычно должны обменяться тремя сегментами (это называется *трехсторонним квитированием*). Как показано на рис. 3.14, эта процедура инициируется вызовом

connect со стороны клиента и завершается, когда сервер получает подтверждение ACK на посланный им сегмент SYN.

Примечание *Возможны, конечно, и другие варианты обмена сегментами. Например, одновременный connect, когда сегменты SYN передаются навстречу друг другу. Но в большинстве случаев соединение устанавливается именно так, как показано на рис. 3.14.*

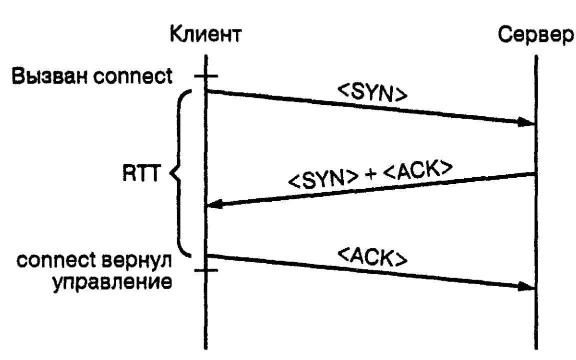


Рис. 3.14 Обычная процедура трехстороннего квитирования

При использовании блокирующего сокета вызов connect не возвращает управления, пока не придет подтверждение ACK на посланный клиентом SYN. Поскольку для этого требуется, по меньшей мере, время RTT, а при перегрузке сети или недоступности хоста на другом конце – даже больше, часто бывает полезно прервать вызов connect. Обычно TCP делает это самостоятельно, но время ожидания (как правило, 75 с) может быть слишком велико для приложения. В некоторых реализациях, например в системе Solaris, есть опции сокета для управления величиной тайм-аута connect, но, к сожалению, они имеются не во всех системах.

Использование вызова alarm

Есть два способа прерывания connect по тайм-ауту. Самый простой – окружить этот вызов обращениями к alarm. Предположим, например, что вы не хотите ждать завершения connect более пяти секунд. Тогда можно модифицировать каркас tcpclient.skel (листинг 2.6), добавив простой обработчик сигнала и немного видоизменив функцию main:

```
void alarm_hndlr( int sig )
{
   return;
}

int main( int argc, char **argv )
{
   ...
   signal( SIGALRM, alarm_hndlr );
   alarm( 5 );
```

```
    rc = connect( s, ( struct sockaddr * )&peer, sizeof( peer ) );
    alarm( 0 );
    if ( rc < 0 )
    {
       if ( errno == EINTR )
          error( 1, 0, "истек тайм-аут connect\n" );
    ...
}
```

Назовем программу, созданную по этому каркасу, connecto и попытаемся с ее помощью соединиться с очень загруженным Web-сервером Yahoo. Получится ожидаемый результат:

```
bsd: $ connectto yahoo.com daytime
connectto: истек тайм-аут connect      спустя 5 с
bsd: $
```

Хотя это и простое решение, с ним связано две потенциальных проблемы. Сначала обсудим их, а потом рассмотрим другой метод – он сложнее, но лишен этих недостатков.

Прежде всего в данном примере подразумевается, что «тревожный» таймер, используемый в вызове alarm, нигде в программе не применяется, и, значит, для сигнала SIGALRM не установлен другой обработчик. Если таймер уже взведен где-то еще, то приведенный код его переустановит, поэтому старый таймер не сработает. Правильнее было бы сохранить и затем восстановить время, оставшееся до срабатывания текущего таймера (его возвращает вызов alarm), а также сохранить и восстановить текущий обработчик сигнала SIGALRM (его адрес возвращает вызов signal). Чтобы все было корректно, надо было также получить время, проведенное в вызове connect, и вычесть его из времени, оставшегося до срабатывания исходного таймера.

Далее, для упрощения вы завершаете клиент, если connect не вернул управления вовремя. Вероятно, нужно было бы предпринять иные действия. Однако надо иметь в виду, что перезапустить connect нельзя. Дело в том, что в результате вызова connect сокет остался привязанным к ранее указанному адресу, так что попытка повторного выполнения приведет к ошибке «Address already in use». При желании повторить connect, возможно, немного подождав, придется сначала закрыть, а затем заново открыть сокет, вызвав close (или closesocket) и socket.

Еще одна потенциальная проблема в том, что некоторые UNIX-системы могут автоматически возобновлять вызов connect после возврата из обработчика сигнала. В таком случае connect не вернет управления, пока не истечет тайм-аут TCP. Во всех современных вариантах системы UNIX поддерживается вызов sigaction, который можно использовать вместо signal. В таком случае следует указать, хотите ли вы рестартовать connect. Но в некоторых устаревших версиях UNIX этот вызов не поддерживается, и тогда использование alarm для прерывания connect по тайм-ауту затруднительно.

Если нужно вывести всего лишь диагностическое сообщение и завершить сеанс, то это можно сделать в обработчике сигнала. Поскольку это происходит до рестарта connect, не имеет значения, поддерживает система вызов sigaction или нет. Однако если нужно предпринять какие-то другие действия, то, вероятно, придется

выйти из обработчика с помощью функции `longjmp`, а это неизбежно приводит к возникновению гонки.

Примечание *Следует заметить, что гонка возникает и в более простом случае, когда вы завершаете программу. Предположим, что соединение успешно установлено, и `connect` вернул управление. Однако прежде чем вы успели его отменить, таймер сработал, что привело к вызову обработчика сигнала и, следовательно, к завершению программы.*

```
alarm( 5 );
rc = connect( s, NULL, NULL );
/* здесь срабатывает таймер */
alarm ( 0 );
```

Вы завершаете программу, хотя соединение и удалось установить. В первоначальном коде такая гонка не возникает, поскольку даже если таймер сработает между возвратом из `connect` и вызовом `alarm`, обработчик сигнала вернет управление, не предпринимая никаких действий.

Принимая это во внимание, многие эксперты считают, что для прерывания вызова `connect` по тайм-ауту лучше использовать `select`.

Использование select

Другой, более общий метод организации тайм-аута `connect` состоит в том, чтобы сделать сокет неблокирующим, а затем ожидать с помощью вызова `select`. При таком подходе удается избежать большинства трудностей, возникающих при попытке воспользоваться `alarm`, но остаются проблемы переносимости даже между разными UNIX-системами.

Сначала рассмотрим код установления соединения. В каркасе `tcpclient.skel` модифицируйте функцию `main`, как показано в листинге 3.25.

Листинг 3.25. Прерывание connect по тайм-ауту с помощью select

connectto1.c

```
int main( int argc, char **argv )
{
    fd_set rdevents;
    fd_set wrevents;
    fd_set exevents;
    struct sockaddr_in peer;
    struct timeval tv;
    SOCKET s;
    int flags;
    int rc;

    INIT();

    set_address( argv[ 1 ], argv[ 2 ], &peer, "tcp" );
```

```
13     s = socket( AF_INET, SOCK_STREAM, 0 );
14     if ( !isvalidsock( s ) )
15        error( 1, errno, "ошибка вызова socket " );
16     if( ( flags = fcntl( s, F_GETFL, 0 ) ) < 0 )
17        error( 1, errno, "ошибка вызова fcntl (F_GETFL)" );
18     if ( fcntl( s, F_SETFL, flags | O_NONBLOCK ) < 0 )
19        error( 1, errno, "ошибка вызова fcntl (F_SETFL)" );
20     if ( ( rc = connect( s, ( struct sockaddr * )&peer,
21         sizeof( peer ) ) ) && errno != EINPROGRESS )
22        error( 1, errno, "ошибка вызова connect" );
23     if ( rc == 0 )  /* Уже соединен? */
24     {
25        if ( fcntl( s, F_SETFL, flags ) < 0 )
26        error( 1, errno, "ошибка вызова fcntl (восстановление флагов)"
);
27        client( s, &peer );
28        EXIT( 0 );
29     }
30     FD_ZERO( &rdevents );
31     FD_SET( s, &rdevents );
32     wrevents = rdevents;
33     exevents = rdevents;
34     tv.tv_sec = 5;
35     tv.tv_usec = 0;
36     rc = select( s + 1, &rdevents, &wrevents, &exevents, &tv );
37     if ( rc < 0 )
38        error( 1, errno, "ошибка вызова select" );
39     else if ( rc == 0 )
40        error( 1, 0, "истек тайм-аут connect\n" );
41     else if ( isconnected( s, &rdevents, &wrevents, &exevents ) )
42     {
43        if ( fcntl( s, F_SETFL, flags ) < 0 )
44        error( 1, errno, "ошибка вызова fcntl (восстановление флагов)"
);
45        client( s, &peer );
46     }
47     else
48        error( 1, errno, "ошибка вызова connect" );
49     EXIT( 0 );
50 }
```

connectto1.c

Инициализация

16-19 Получаем текущие флаги, установленные для сокета, с помощью операции OR, добавляем к ним флаг O_NONBLOCK и устанавливаем новые флаги.

Инициирование connect

20-29 Начинаем установление соединения с помощью вызова `connect`. Поскольку сокет помечен как неблокирующий, `connect` немедленно

возвращает управление. Если соединение уже установлено (это возможно, если, например, вы соединялись с той машиной, на которой запущена программа), то `connect` вернет нуль, поэтому возвращаем сокет в режим блокирования и вызываем функцию `client`. Обычно в момент возврата из `connect` соединение еще не установлено, и приходит код `EINPROGRESS`. Если возвращается другой код, то печатаем диагностическое сообщение и завершаем программу.

Вызов select

30-36 Подготавливаем, как обычно, данные для `select` и, в частности, устанавливаем тайм-аут на пять секунд. Также следует объявить заинтересованность в событиях исключения. Зачем – станет ясно позже.

Обработка код возврата select

37-40 Если `select` возвращает код ошибки или признак завершения по тайм-ауту, то выводим сообщение и заканчиваем работу. В случае ответа можно было бы, конечно, сделать что-то другое.

41-46 Вызываем функцию `isconnected`, чтобы проверить, удалось ли установить соединение. Если да, возвращаем сокет в режим блокирования и вызываем функцию `client`. Текст функции `isconnected` приведен в листингах 3.26 и 3.27.

47-48 Если соединение не установлено, выводим сообщение и завершаем сеанс.

К сожалению, в UNIX и в Windows применяются разные методы уведомления об успешной попытке соединения. Поэтому проверка вынесена в отдельную функцию. Сначала приводится UNIX-версия функции `isconnected`.

В UNIX, если соединение установлено, сокет доступен для записи. Если же произошла ошибка, то сокет будет доступен одновременно для записи и для чтения. Однако на это нельзя полагаться при проверке успешности соединения, поскольку можно возвратиться из `connect` и получить первые данные еще до обращения к `select`. В таком случае сокет будет доступен и для чтения, и для записи – в точности, как при возникновении ошибки.

Листинг 3.26. UNIX-версия функции isconnected

connectto1.c

```
int isconnected( SOCKET s, fd_set *rd, fd_set *wr, fd_set *ex )
{
    int err;
    int len = sizeof( err );

    errno = 0;          /* Предполагаем, что ошибки нет. */
    if ( !FD_ISSET( s, rd ) && !FD_ISSET( s, wr ) )
        return 0;

    if ( getsockopt( s, SOL_SOCKET, SO_ERROR, &err, &len ) < 0 )
        return 0;
    errno = err;    /* Если мы не соединились. */
    return err == 0;
}
```

connectto1.c

5-7 Если сокет не доступен ни для чтения, ни для записи, значит, соединение не установлено, и возвращается нуль. Значение errno заранее установлено в нуль, чтобы вызывающая программа могла определить, что сокет, действительно, не готов (разбираемый случай) или имеет место ошибка.

8-11 Вызываем getsockopt для получения статуса сокета. В некоторых версиях UNIX getsockopt возвращает в случае ошибки –1. В таком случае записываем в errno код ошибки. В других версиях система просто возвращает статус, оставляя его проверку пользователю. Идея кода, который корректно работает в обоих случаях, позаимствована из книги [Stevens 1998].

Согласно спецификации Winsock, ошибки, которые возвращает connect через неблокирующий сокет, индицируются путем возбуждения события исключения в select. Следует заметить, что в UNIX событие исключения всегда свидетельствует о поступлении срочных данных. Версия функции isconnected для Windows показана в листинге 3.27.

Листинг 3.27. Windows-версия функции isconnected

```
                                                          connectto1.c
1 int isconnected( SOCKET s, fd_set *rd, fd_set *wr, fd_set *ex )
2 {
3    WSASetLastError( 0 );
4    if ( !FD_ISSET( s, rd ) && !FD_ISSET( s, wr ) )
5       return 0;
6    if ( FD_ISSET( s, ex ) )
7       return 0;
8    return 1;
9 }
                                                          connectto1.c
```

3-5 Так же, как и в версии для UNIX, проверяем, соединен ли сокет. Если нет, устанавливаем последнюю ошибку в нуль и возвращаем нуль.

6-8 Если для сокета есть событие исключения, возвращается нуль, в противном случае – единица.

Резюме

Как видите, для переноса на разные платформы прерывать вызов connect с помощью тайм-аута более сложно, чем обычно. Поэтому при выполнении такого действия надо уделить особое внимание платформе.

Наконец, следует понимать, что сократить время ожидания connect можно, а увеличить – нет. Все вышерассмотренные методы направлены на то, чтобы прервать вызов connect *раньше*, чем это сделает TCP. Не существует переносимого механизма для изменения значения тайм-аута TCP на уровне одного сокета.

Совет 26. Избегайте копирования данных

Во многих сетевых приложениях, занимающихся, прежде всего, переносом данных между машинами, большая часть времени процессора уходит на копирование

данных из одного буфера в другой. В этом разделе будет рассмотрено несколько способов уменьшения объема копирования, что позволит «бесплатно» повысить производительность приложения. Предложение избегать копирования больших объемов данных в памяти оказывается не таким революционным, поскольку именно так всегда и происходит. Массивы передаются не целиком, используются только указатели на них.

Конечно, обычно данные между функциями, работающими внутри одного процесса, не копируются. Но в многопроцессных приложениях часто приходится передавать большие объемы данных от одного процесса другому с помощью того или иного механизма межпроцессного взаимодействия. И даже в рамках одного процесса часто доводится заниматься копированием, если сообщение состоит более чем из двух частей, которые нужно объединить перед отправкой другому процессу или другой машине. Типичный пример такого рода, обсуждавшийся в совете 24, – это добавление заголовка в начало сообщения. Сначала копируется в буфер заголовок, а вслед за ним – само сообщение.

Стремление избегать копирования данных внутри одного процесса – признак хорошего стиля программирования. Если заранее известно, что сообщению будет предшествовать заголовок, то надо оставить для него место в буфере. Иными словами, если ожидается заголовок, описываемый структурой `struct hdr`, то прочитать данные можно было бы так:

```
rc = read( fd, buf + sizeof( struct hdr ) ),
   sizeof( buf ) - sizeof( struct hdr );
```

Пример применения такой техники содержится в листинге 3.6.

Еще один прием – определить пакет сообщения в виде структуры, одним из элементов которой является заголовок. Тогда можно просто прочитать заголовок в одном из полей:

```
struct {
   struct hdr header; /* Структура определена в другом месте. */
   char data[ DATASZ ];
} packet;
rc = read( fd, packet, data. sizeof( packet data ) );
```

Пример использования этого способа был продемонстрирован в листинге 2.15. Там же говорилось, что при определении такой структуры следует проявлять осмотрительность.

Третий, очень гибкий, прием заключается в применении операции записи со сбором – листинги 3.23 (UNIX) и 3.24 (Winsock). Он позволяет объединять части сообщения с различными размерами.

Избежать копирования данных намного труднее, когда есть несколько процессов. Эта проблема часто возникает в системе UNIX, где многопроцессные приложения – распространенная парадигма (рис. 3.4). Обычно в этой ситуации проблема даже острее, так как механизмы IPC, как правило, копируют данные отправляющего процесса в пространство ядра, а затем из ядра в пространство принимающего процесса, то есть копирование происходит дважды. Поэтому необходимо применять хотя бы один из вышеупомянутых методов, чтобы избежать лишних операций копирования.

Буферы в разделяемой памяти

Обойтись почти без копирования, даже между разными процессами, можно, воспользовавшись разделяемой памятью. Разделяемая память – это область памяти, доступная сразу нескольким процессам. Каждый процесс отображает блок виртуальной памяти на адрес в собственном адресном пространстве (в разных процессах эти адреса могут быть различны), а затем обращается к нему, как к собственной памяти.

Идея состоит в том, чтобы создать массив буферов в разделяемой памяти, построить сообщение в одном из них, а затем передать индекс буфера следующему процессу, применяя механизм IPC. При этом «перемещается» только одно целое число, представляющее индекс буфера в массиве. Например, на рис. 3.15 в качестве механизма IPC используется TCP для передачи числа 3 от процесса 1 процессу 2. Когда процесс 2 получает это число, он определяет, что приготовлены данные в буфере `smbarray[ 3 ]`.

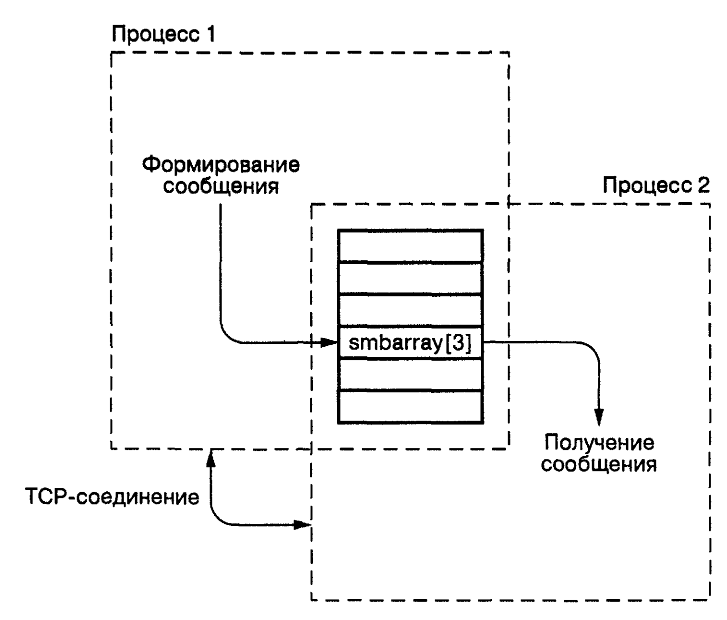


Рис. 3.15. Передача сообщений через буфер в разделяемой памяти

На рис. 3.15 два пунктирных прямоугольника представляют адресные пространства процессов 1 и 2, а их пересечение – общий сегмент разделяемой памяти, который каждый из процессов отобразил на собственное адресное пространство. Массив буферов находится в разделяемом сегменте и доступен обоим процессам. Процесс 1 использует отдельный канал IPC (в данном случае – TCP) для информирования процесса 2 о том, что для него готовы данные, а также место, где их искать.

Хотя здесь показано только два процесса, этот прием прекрасно работает для любого их количества. Кроме того, процесс 2, в свою очередь, может передать сообщение процессу 1, получив буфер в разделяемой памяти, построив в нем сообщение и послав процессу 1 индекс буфера в массиве.

Единственное, что пока отсутствует, – это синхронизация доступа к буферам, то есть предотвращение ситуации, когда два процесса одновременно получат один и тот же буфер. Это легко делается с помощью мьютекса, что и будет продемонстрировано ниже.

Система буферов в разделяемой памяти

Описанную систему буферов в разделяемой памяти легко реализовать. Основная сложность в том, как получить область разделяемой памяти, отобразить ее на собственное адресное пространство и синхронизировать доступ к буферам. Конечно, это зависит от конкретной системы, поэтому далее будет приведена реализация как для UNIX, так и для Windows.

Но прежде чем перейти к системно-зависимым частям, обратимся к API и его реализации. На пользовательском уровне система состоит из пяти функций:

```
#include "etcp.h"

void init_smb( int init_freelist);

void *smballoc( void );
```

Возвращаемое значение: указатель на буфер в разделяемой памяти.

```
void smbfree( void *smbptr );

void smbsend( SOCKET s, void * smbptr );

void *smbrecv( SOCKET s );
```

Возвращаемое значение: указатель на буфер в разделяемой памяти.

Перед тем как пользоваться системой, каждый процесс должен вызвать функцию `init_smb` для получения и инициализации области разделяемой памяти и синхронизирующего мьютекса. При этом только один процесс должен вызвать `init_smb` с параметром `init_freelist`, равным `TRUE`.

Для получения буфера в разделяемой памяти служит функция `smballoc`, возвращающая указатель на только что выделенный буфер. Когда буфер уже не нужен, процесс может вернуть его системе с помощью функции `smb_free`.

Построив сообщение в буфере разделяемой памяти, процесс может передать буфер другому процессу, вызвав `smbsend`. Как уже говорилось, при этом передается только индекс буфера в массиве. Для получения буфера от отправителя процесс-получатель вызывает функцию `smbrecv`, которая возвращает указатель на буфер.

В данной системе для передачи индексов буферов используется TCP в качестве механизма IPC, но это не единственное и даже не оптимальное решение. Так удобнее, поскольку этот механизм работает как под UNIX, так и под Windows, и к тому же можно воспользоваться уже имеющимися средствами, а не изучать другие методы IPC. В системе UNIX можно было бы применить также сокеты в адресном домене UNIX или именованные каналы. В Windows доступны `SendMessage`, `QueueUserAPC` и именованные каналы.

Начнем рассмотрение реализации с функций `smballoc` и `smbfree` (листинг 3.28).

Листинг 3.28. Функции smballoc и smbfree

smb.c

```
 1 #include "etcp.h"
 2 #define FREE_LIST    smbarray[ NSMB ].nexti
 3 typedef union
 4 {
 5    int nexti;
 6    char buf[ SMBUFSZ ];
 7 } smb_t;
 8 smb_t *smbarray;
 9 void *smballoc( void )
10 {
11    smb_t *bp;
12    lock_buf();
13    if ( FREE_LIST < 0 )
14    error( 1, 0, "больше нет буферов в разделяемой памяти\n" );
15    bp = smbarray + FREE_LIST;
16    FREE_LIST  = bp->nexti;
17    unlock_buf();
18    return bp;
19 }
20 void smbfree( void *b )
21 {
22    smb_t *bp;
23    bp = b;
24    lock_buf();
25    bp->nexti  = FREE_LIST;
26    FREE_LIST  = bp - smbarray;
27    unlock_buf();
28 }
```

smb.c

Заголовок

2-8 Доступные буфера хранятся в списке свободных. При этом в первых `sizeof( int )` байтах буфера хранится индекс следующего свободного буфера. Такая организация памяти отражена в объединении `smb_t`. В конце массива буферов есть одно целое число, которое содержит либо индекс первого буфера в списке свободных, либо –1, если этот список пуст. Доступ к этому числу вы получаете, адресуя его как `smbarray [ NSMB ].nexti`. Для удобства это выражение инкапсулировано в макрос `FREE_LIST`. На сам массив буферов указывает переменная `smbarray`. Это, по сути, указатель на область разделяемой памяти, которую каждый процесс отображает на свое адресное пространство. В массиве использованы индексы, а не адреса элементов, так как в разных процессах эти адреса могут быть различны.

smballoc

12 Вызываем функцию `lock_buf`, чтобы другой процесс не мог обратиться к списку свободных. Реализация этой функции зависит от системы. В UNIX будут использованы семафоры, а в Windows – мьютексы.

13-16 Получаем буфер из списка свободных. Если больше буферов нет, то выводим диагностическое сообщение и завершаем сеанс. Вместо этого можно было бы вернуть `NULL`.

17-18 Открываем доступ к списку свободных и возвращаем указатель на буфер.

smbfree

23-27 После блокировки списка свободных, возвращаем буфер, помещая его индекс в начало списка. Затем разблокируем список свободных и возвращаем управление.

Далее рассмотрим функции `smbsend` и `smbrecv` (листинг 3.29). Они посылают и принимают целочисленный индекс буфера, которым обмениваются процессы. Эти функции несложно адаптировать под иной механизм межпроцессного взаимодействия.

Листинг 3.29. Функции smbsend и smbrecv

smb.c

```
 1 void smbsend( SOCKET s, void *b )
 2 {
 3    int index;

 4    index = ( smb_t * )b - smbarray;
 5    if ( send( s, ( char * )&index, sizeof( index ), 0 ) < 0 )
 6       error( 1, errno, "smbsend: ошибка вызова send" );
 7 }

 8 void *smbrecv( SOCKET s )
 9 {
10    int index;
11    int rc;

12    rc = readn( s, ( char * )&index, sizeof( index ) );
13    if ( rc == 0 )
14       error( 1, 0, "smbrecv: другой конец отсоединился\n" );
15    else if ( rc != sizeof( index ) )
16       error( 1, errno, "smbrecv: ошибка вызова readn" );
17    return smbarray + index;
18 }
```

smb.c

smbsend

4-6 Вычисляем индекс буфера, на который указывает *b*, и посылаем его другому процессу с помощью `send`.

smbrecv

12-16 Вызываем readn для чтения переданного индекса буфера. В случае ошибки чтения или при получении неожиданного числа байт, выводим сообщение и завершаем работу.

17 В противном случае преобразуем индекс буфера в указатель на него и возвращаем этот указатель вызывающей программе.

Реализация в UNIX

Для завершения реализации системы буферов в разделяемой памяти нужны еще два компонента. Это способ выделения блока разделяемой памяти и отображения его на адресное пространство процесса, а также механизм синхронизации для предотвращения одновременного доступа к списку свободных. Для работы с разделяемой памятью следует воспользоваться механизмом, разработанным в свое время для версии SysV. Можно было бы вместо него применить отображенный на память файл, как в Windows. Кроме того, есть еще разделяемая память в стандарте POSIX – для систем, которые ее поддерживают.

Для работы с разделяемой памятью SysV понадобятся только два системных вызова:

```
#include <sys/shm.h>

int shmget( key_t key, size_t size, int flags );
```

Возвращаемое значение: идентификатор сегмента разделяемой памяти в случае успеха, –1 – в случае ошибки.

```
void shmat( int segid, const void *baseaddr, int flags );
```

Возвращаемое значение: базовый адрес сегмента в случае успеха, –1 – в случае ошибки.

Системный вызов shmget применяется для выделения сегмента разделяемой памяти. Первый параметр, *key*, – это глобальный для всей системы уникальный идентификатор, сегмента. Сегмент будет идентифицироваться целым числом, представление которого в коде ASCII равно SMBM.

Примечание *Использование пространства имен, отличного от файловой системы, считается одним из основных недостатков механизмов IPC, появившихся еще в системе SysV. Для отображения имени файла на ключ IPC можно применить функцию* ftok, *но это отображение не будет уникальным. Кроме того, как отмечается в книге [Stevens 1999], описанная в стандарте SVR4 функция* ftok *дает коллизию (то есть два имени файла отображаются на один и тот же ключ) с вероятностью 75%.*

Параметр *size* задает размер сегмента в байтах. Во многих UNIX-системах его значение округляется до величины, кратной размеру страницы. Параметр *flags* задает права доступа и другие атрибуты сегмента. Значения SHM_R и SHM_W

определяют соответственно права на чтение и на запись для владельца. Права для группы и для всех получают путем сдвига этих значений вправо на три (для группы) или шесть (для всех) бит. Иными словами, право на запись для группы – это `SHM_W >> 3`, а право на чтение для всех – `SHM_R >> 6`. Когда в параметр *flags* с помощью побитовой операции OR включается флаг `IPC_CREATE`, создается сегмент, если раньше его не было. При дополнительном включении флага `IPC_EXCL` `shmget` вернет код ошибки `EEXIST`, если сегмент уже существует.

Вызов `shmget` только *создает* сегмент в разделяемой памяти. Для отображения его в адресное пространство процесса нужно вызвать `shmat`. Параметр *segid* – это идентификатор сегмента, который вернул вызов `shmget`. При желании можно указать адрес *baseaddr*, на который ядро должно отобразить сегмент, но обычно этот параметр оставляют равным `NULL`, позволяя ядру самостоятельно выбрать адрес. Параметр *flags* используется, если значение *baseaddr* не равно `NULL`, – он управляет выравниваем заданного адреса на приемлемую для ядра границу.

Для построения механизма взаимного исключения следует воспользоваться SysV-семафорами. Хотя они небезупречны (в частности, им присуща та же проблема нового пространства имен, что и разделяемой памяти), SysV-семафоры широко используются в современных UNIX-системах и, следовательно, обеспечивают максимальную переносимость. Как и в случае разделяемой памяти, сначала надо получить и инициализировать семафор, а потом уже его применять. В данной ситуации понадобятся три относящихся к семафорам системных вызова.

Вызов `semget` аналогичен `shmget`: он получает у операционной системы семафор и возвращает его идентификатор. Параметр *key* имеет тот же смысл, что и для `shmget` – он именует семафор. В SysV-семафоры выделяются группами, и параметр *nsems* означает, сколько семафоров должно быть в запрашиваемой группе. Параметр *flags* такой же, как для `shmget`.

```
#include <sys/sem.h>

int semget( key_t key, int nsems, int flags );
```

Возвращаемое значение: идентификатор семафора в случае успеха, –1 – в случае ошибки.

```
int semctl( int semid, int semnum, int cmd, ... );
```

Возвращаемое значение: неотрицательное число в случае успеха, –1 – в случае ошибки.

```
int semop( int semid, struct sembuf *oparray, size_t nops );
```

Возвращаемое значение: 0 в случае успеха, –1 – в случае ошибки.

Здесь использована `semctl` для задания начального значения семафора. Этот вызов служит также для установки и получения различных управляющих параметров, связанных с семафором. Параметр *semid* – это идентификатор семафора, возвращенный вызовом `semget`. Параметр *semnum* означает конкретный семафор из группы. Поскольку будет выделяться только один семафор, значение этого параметра всегда равно нулю. Параметр *cmd* – это код выполняемой операции.

У вызова semget могут быть и дополнительные параметры, о чем свидетельствует многоточие в прототипе.

Вызов semop используется для увеличения или уменьшения значения семафора. Когда процесс пытается уменьшить семафор до отрицательного значения, он переводится в состояние ожидания, пока другой процесс не увеличит семафор до значения, большего или равного тому, на которое первый процесс пытался его уменьшить. Поскольку надо использовать семафоры в качестве мьютексов, следует уменьшать значение на единицу для блокировки списка свободных и увеличивать на единицу – для разблокировки. Так как начальное значение семафора равно единице, в результате процесс, пытающийся заблокировать уже блокированный список свободных, будет приостановлен.

Параметр *semid* – это идентификатор семафора, возвращенный semget. Параметр *oparray* указывает на массив структур *sembuf*, в котором заданы операции над одним или несколькими семафорами из группы. Параметр *nops* задает число элементов в массиве *oparray*.

Показанная ниже структура sembuf содержит информацию о том, к какому семафору применить операцию (sem_num), увеличить или уменьшить значение семафора (sem_op), а также флаг для двух специальных действий (sem_flg):

```
struct sembuf {
   u_short sem_num;    /* Номер семафора. */
   short sem_op;       /* Операция над семафором. */
   short sem_flg;      /* Флаги операций. */
};
```

В поле sem_flg могут быть подняты два бита флагов:

- IPC_NOWAIT – означает, что semop должна вернуть код EAGAIN, а не приостанавливать процесс, если в результате операции значение семафора окажется отрицательным;
- SEM_UNDO – означает, что semop должна отменить действие всех операций над семафором, если процесс завершается, то есть мьютекс будет освобожден.

Теперь рассмотрим UNIX-зависимую часть кода системы буферов в разделяемой памяти (листинг 3.30).

Листинг 3.30. Функция init_smb для UNIX

smb.c

```
#include <sys/shm.h>
#include <sys/sem.h>
#define MUTEX_KEY 0x534d4253  /* SMBS */
#define SM_KEY 0x534d424d     /* SMBM */
#define lock_buf() if ( semop( mutex, &lkbuf, 1 ) < 0 ) \
                error( 1, errno, "ошибка вызова semop" )
#define unlock_buf()  if ( semop( mutex, &unlkbuf, 1 ) < 0 ) \
                error( 1, errno, "ошибка вызова semop" )
int mutex;
```

```
struct sembuf lkbuf;
struct sembuf unlkbuf;

void init_smb( int init_freelist )
{
   union semun arg;
   int smid;
   int i;
   int rc;

   lkbuf.sem_op = -1;
   lkbuf.sem_flg = SEM_UNDO;
   unlkbuf.sem_op = 1;
   unlkbuf.sem_flg = SEM_UNDO;
   mutex = semget( MUTEX_KEY, 1,
      IPC_EXCL | IPC_CREAT | SEM_R | SEM_A );
   if ( mutex >= 0 )
   {
      arg.val = 1;
      rc = semctl( mutex, 0, SETVAL, arg );
      if ( rc < 0 )
         error( 1, errno, "semctl failed" );
   }
   else if ( errno == EEXIST )
   {
      mutex = semget( MUTEX_KEY, 1, SEM_R | SEM_A );
      if ( mutex < 0 )
         error( 1, errno, "ошибка вызова semctl" );
   }
   else
      error( 1, errno, "ошибка вызова semctl" );

   smid = shmget( SM_KEY, NSMB * sizeof( smb_t ) + sizeof( int ),
      SHM_R | SHM_W | IPC_CREAT );
   if ( smid < 0 )
      error( 1, errno, "ошибка вызова shmget" );
   smbarray = ( smb_t * )shmat( smid, NULL, 0 );
   if ( smbarray == ( void * )-1 )
      error( 1, errno, "ошибка вызова shmat" );

   if ( init_freelist )
   {
      for ( i = 0; i < NSMB - 1; i++ )
         smbarray[ i ].nexti = i + 1;
      smbarray[ NSMB - 1 ].nexti = -1;
      FREE_LIST = 0;
   }
}
```

smb.c

Макросы и глобальные переменные

3-4 Определяем ключи сегмента разделяемой памяти (SMBM) и семафора (SMBS).

5-8 Определяем примитивы блокировки и разблокировки в терминах операций над семафорами.

9-11 Объявляем переменные для семафоров, используемых для реализации мьютекса.

Получение и инициализация семафора

18-21 Инициализируем операции над семафорами, которыми будем пользоваться для блокировки и разблокировки списка свободных.

22-38 Этот код создает и инициализирует семафор. Вызываем `semget` с флагами `IPC_EXCL` и `IPC_CREAT`. В результате семафор будет создан, если он еще не существует, и в этом случае `semget` вернет идентификатор семафора, который инициализируем единицей (разблокированное состояние). Если же семафор уже есть, то снова вызываем `semget`, уже не задавая флагов `IPC_EXCL` и `IPC_CREAT`, для получения идентификатора этого семафора. Как отмечено в книге [Stevens 1999], теоретически здесь возможна гонка, но не в данном случае, поскольку сервер вызывает `init_smb` перед вызовом `listen`, а клиент не сможет обратиться к нему, пока вызов `connect` не вернет управление.

Примечание *В книге [Stevens 1999] рассматриваются условия, при которых возможна гонка, и показывается, как ее избежать.*

Получение, отображение и инициализация буферов в разделяемой памяти

39-45 Выделяем сегмент разделяемой памяти и отображаем его на свое адресное пространство. Если сегмент уже существует, то `shmget` возвращает его идентификатор.

46-53 Если `init_smb` была вызвана с параметром *`init_freelist`*, равным `TRUE`, то помещаем все выделенные буферы в список свободных и возвращаем управление.

Реализация в Windows

Прежде чем демонстрировать систему в действии, рассмотрим реализацию для Windows. Как было упомянуто выше, весь системно-зависимый код сосредоточен в функции `init_smb`. В Windows мьютекс создается очень просто – достаточно вызвать функцию `CreateMutex`.

```
#include <windows.h>

HANDLE CreateMutex(  LPSECURITY_ATTRIBUTES lpsa,
            BOOL fInitialOwner, LPTSTR lpszMutexName );
```

Возвращаемое значение: описание мьютекса в случае успеха, NULL – в случае ошибки.

Параметр *lpsa* – это указатель на структуру с атрибутами защиты. Здесь эта возможность не нужна, так что вместо этого аргумента передадим NULL. Параметр *fInitialOwner* означает, будет ли создатель мьютекса его начальным владельцем, то есть следует ли сразу заблокировать мьютекс. Параметр *lpszMutexName* – это имя мьютекса, по которому к нему могут обратиться другие процессы. Если мьютекс уже существует, то CreateMutex просто вернет его описание.

Блокировка и разблокировка мьютекса выполняются соответственно с помощью функций WaitForSingleObject и ReleaseMutex.

```
#include <windows.h>

DWORD WaitForSingleObject( HANDLE hObject, DWORD dwTimeout );
```

Возвращаемое значение: WAIT_OBJECT_0 (0) в случае успеха, ненулевое значение – в случае ошибки.

```
BOOL ReleaseMutex( HANDLE hMutex );
```

Возвращаемое значение: TRUE в случае успеха, FALSE – в случае ошибки.

Параметр *hObject* функции WaitForSingleObject – это описание ожидаемого объекта (в данном случае мьютекса). Если объект, заданный с помощью *hObject*, не занят (signaled), то WaitForSingleObject занимает его и возвращает управление. Если же объект занят (not signaled), то обратившийся поток переводится в состояние ожидания до тех пор, пока объект не освободится. После этого WaitForSingleObject переведет объект в занятое состояние и вернет в работу «спящий» поток. Параметр *dwTimeout* задает время (в миллисекундах), в течение которого потоком ожидается освобождение объекта. Если тайм-аут истечет прежде, чем объект освободится, то WaitForSingleObject вернет код WAIT_TIMEOUT. Таймер можно подавить, задав в качестве *dwTimeout* значение INFINITE.

Когда поток заканчивает работу с критической областью, охраняемой мьютексом, он разблокирует его вызовом ReleaseMutex, передавая описание мьютекса *hMutex* в качестве параметра.

В Windows вы получаете сегмент разделяемой памяти, отображая файл на память каждого процесса, которому нужен доступ к разделяемой памяти (в UNIX есть аналогичный системный вызов mmap). Для этого сначала создается обычный файл с помощью функции CreateFile, затем – отображение файла посредством вызова CreateFileMapping, а уже потом оно отображается на ваше адресное пространство вызовом MapViewOfFile.

Параметр *hFile* в вызове CreateFileMapping – это описание отображаемого файла. Параметр *lpsa* указывает на структуру с атрибутами безопасности, которые в данном случае не нужны. Параметр *fdwProtect* определяет права доступа к объекту в памяти. Он может принимать значения PAGE_READONLY, PAGE_READWRITE или PAGE_WRITECOPY. Последнее значение заставляет ядро сделать отдельную копию данных, если процесс пытается записывать в страницу памяти. Здесь используется PAGE_READWRITE, так как будет производится и чтение, и запись в разделяемую память. Существуют также дополнительные флаги, объединяемые операцией

побитового OR, которые служат для управления кэшированием страниц памяти, но они не понадобятся. Параметры *dwMaximumSizeHigh* и *dwMaximumSizeLow* в совокупности дают 64-разрядный размер объекта в памяти. Параметр *lpszMapName* – это имя объекта. Под данным именем объект известен другим процессам.

```
#include <windows.h>

HANDLE CreateFileMapping( HANDLE hFile, LPSECURITY_ATTRIBUTES lpsa,
                DWORD fdwProtect, DWORD dwMaximumSizeHigh,
                DWORD dwMaximumSizeLow, LPSTR lpszMapName );
```

Возвращаемое значение: описатель отображения файла в случае успеха, NULL – в случае ошибки.

```
LPVOID MapViewOfFile( HANDLE hFileMapObject, DWORD dwDesiredAccess,
                DWORD dwFileOffsetHigh, DWORD dwFileOffsetLow,
                DWORD dwBytesToMap );
```

Возвращаемое значение: адрес, на который отображена память, в случае успеха, NULL – в случае ошибки.

После создания объект в памяти отображается на адресное пространство каждого процесса с помощью функции `MapViewOfFile`. Параметр *hFileMapObj* – это описание, возвращенное после вызова `CreateFileMapping`. Требуемый уровень доступа следует задать с помощью *dwDesiredAccess*. Этот параметр может принимать следующие значения: `FILE_MAP_WRITE` (доступ на чтение и запись), `FILE_MAP_READ` (доступ только на чтение), `FILE_MAP_ALL_ACCESS` (то же, что `FILE_MAP_WRITE`) и `FILE_MAP_COPY`. Если присвоено последнее значение, то при попытке записи создается отдельная копия данных. Параметры *dwFileOffsetHigh* и `dwFileOffsetLow` задают смещение от начала файла, с которого следует начинать отображение. Нужно отобразить файл целиком, поэтому оба параметра будут равны 0. Размер отображаемой области памяти задается с помощью параметра *dwBytesToMap*.

Подробнее использование мьютексов и отображение памяти в Windows рассматриваются в книге [Richter 1997].

Теперь можно представить версию `init_smb` для Windows. Как видно из листинга 3.31, она очень напоминает версию для UNIX.

Листинг 3.31. Функция init_smb для Windows

smb.c

```
#define FILENAME   "./smbfile"
#define lock_buf() if ( WaitForSingleObject( mutex, INFINITE )\
             != WAIT_OBJECT_0 ) \
            error( 1, errno, "ошибка вызова lock_buf " )
#define unlock_buf()  if ( !ReleaseMutex( mutex ) )\
            error( 1, errno, "ошибка вызова unlock_buf" )
HANDLE mutex;
void init_smb( int init_freelist )
```

```
{
   HANDLE hfile;
   HANDLE hmap;
   int i;

   mutex = CreateMutex( NULL, FALSE, "smbmutex" );
   if ( mutex == NULL )
      error( 1, errno, "ошибка вызова CreateMutex" );
   hfile = CreateFile( FILENAME,
      GENERIC_READ | GENERIC_WRITE,
      FILE_SHARE_READ | FILE_SHARE_WRITE,
      NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL );
   if ( hfile == INVALID_HANDLE_VALUE )
      error( 1, errno, "ошибка вызова CreateFile" );
   hmap = CreateFileMapping( hfile, NULL, PAGE_READWRITE,
      0, NSMB * sizeof( smb_t ) + sizeof( int ), "smbarray" );
   smbarray = MapViewOfFile( hmap, FILE_MAP_WRITE, 0, 0, 0 );
   if ( smbarray == NULL )
      error( 1, errno, "ошибка вызова MapViewOfFile" );

   if ( init_freelist )
   {
      for ( i = 0; i < NSMB - 1; i++ )
         smbarray[ i ].nexti = i + 1;
      smbarray[ NSMB - 1 ].nexti = -1;
      FREE_LIST = 0;
   }
}
```

smb.c

Для тестирования всей системы следует написать небольшие программы клиентской (листинг 3.32) и серверной (листинг 3.33) частей.

Листинг 3.32. Клиент, использующий систему буферов в разделяемой памяти

smbc.c

```
#include "etcp.h"

int main( int argc, char **argv )
{
   char *bp;
   SOCKET s;

   INIT();
   s = tcp_client( argv[ 1 ], argv[ 2 ] );
   init_smb( FALSE );
   bp = smballoc();
   while ( fgets( bp, SMBUFSZ, stdin ) != NULL  )
   {
      smbsend( s, bp );
      bp = smballoc();
```

```
}
EXIT( 0 );
}
```

smbc.c

Листинг 3.33. Сервер, использующий систему буферов в разделяемой памяти

smbs.c

```
#include "etcp.h"

int main( int argc, char **argv )
{
   char *bp;
   SOCKET s;
   SOCKET s1;

   INIT();
   init_smb( TRUE );
   s = tcp_server( NULL, argv[ 1 ] );
   s1 = accept( s, NULL, NULL );
   if ( !isvalidsock( s1 ) )
      error( 1, errno, "ошибка вызова accept" );
   for ( ;; )
   {
      bp = smbrecv( s1 );
      fputs( bp, stdout );
      smbfree( bp );
   }
   EXIT( 0 );
}
```

smbs.c

Запустив эти программы, получите ожидаемый результат:

```
bsd: $ smbc localhost 9000
Hello
World!
^C
bsd: $
```

```
bsd: $ smbs 9000
Hello
Wolds!
smbs: smbrecv: другой конец отсоединился
bsd: $
```

Обратите внимание, что smbc читает каждую строку из стандартного ввода прямо в буфер в разделяемой памяти, а smbs копирует каждую строку из буфера сразу на стандартный вывод, поэтому не возникает лишнего копирования данных.

Резюме

В этом разделе описано, как избежать ненужного копирования данных. Во многих сетевых приложениях на копирование данных из одного буфера в другой тратится большая часть времени процессора.

Разработана схема взаимодействия между процессами, в которой используется система буферов в разделяемой памяти. Это позволило передавать единственный экземпляр данных от одного процесса другому. Такая схема работает и в UNIX, и в Windows.

Совет 27. Обнуляйте структуру sockaddr_in

Хотя обычно используется только три поля из структуры sockaddr_in: sin_family, sin_port и sin_addr, но, как правило, в ней есть и другие поля. Например, во многих реализациях есть поле sin_len, содержащее длину структуры. В частности, оно присутствует в системах, производных от версии 4.3BSD Reno и более поздних. Напротив, в спецификации Winsock этого поля нет.

Если сравнить структуры sockaddr_in в системе FreeBSD

```
struct sockaddr_in {
    u_char sin_len;
    u_char sin_family;
    u_char sin_port;
    struct in_addr sin_addr;
    char sin_zero[8];
};
```

и в Windows

```
struct sockaddr_in {
    short    sin_family;
    u_short  sin_port;
    struct in_addr sin_addr;
    char     sin_zero[8];
};
```

то видно, что в обеих структурах есть дополнительное поле sin_zero. Хотя это поле и не используется (оно нужно для того, чтобы длина структуры sockaddr_in была равна в точности 16 байт), но тем не менее должно быть заполнено нулями.

Примечание *Причина в том, что в некоторых реализациях во время привязки адреса к сокету производится двоичное сравнение этой адресной структуры с адресами каждого интерфейса. Такой код будет работать только в том случае, если поле* sin_zero *заполнено нулями.*

Поскольку в любом случае необходимо обнулить поле sin_zero, обычно перед использованием адресной структуры ее полностью обнуляют. В этом случае заодно очищаются и все дополнительные поля, так что не будет проблем из-за недокументированных полей. Посмотрите на листинг 2.3 – сначала в функции set_address делается вызов bzero для очистки структуры sockaddr_in.

Совет 28. Не забывайте о порядке байтов

В современных компьютерах целые числа хранятся по-разному, в зависимости от архитектуры. Рассмотрим 32-разрядное число 305419896 (0x12345678). Четыре байта этого числа могут храниться двумя способами: сначала два старших байта (такой порядок называется тупоконечным – big endian)

```
12 34 56 78
```

или сначала два младших байта (такой порядок называется остроконечным – little endian)

```
78 56 34 12
```

Примечание *Термины «тупоконечный» и «остроконечный» ввел Коэн [Cohen 1981], считавший, что споры о том, какой формат лучше, сродни распрям лилипутов из романа Свифта «Путешествия Гулливера», которые вели бесконечные войны, не сумев договориться, с какого конца следует разбивать яйцо – с тупого или острого. Раньше были в ходу и другие форматы, но практически во всех современных машинах применяется либо тупоконечный, либо остроконечный порядок байтов.*

Определить формат, применяемый в конкретной машине, можно с помощью следующей текстовой программы, показывающей, как хранится число 0x12345678 (листинг 3.34).

Листинг 3.34. Программа для определения порядка байтов

endian.c

```c
#include <stdio.h>
#include <sys/types.h>
#include "etcp.h"
int main( void )
{
   u_int32_t x = 0x12345678;   /* 305419896 */
   unsigned char *xp = ( char * )&x;

   printf( "%0x %0x %0x %0x\n",
      xp[ 0 ], xp[ 1 ], xp[ 2 ], xp[ 3 ] );
   exit( 0 );
}
```

endian.c

Если запустить эту программу на компьютере с процессором Intel, то получится:

```
bsd: $ endian
78 56 34 12
bsd: $
```

Отсюда ясно видно, это – остроконечная архитектура.

Конкретный формат хранения иногда в шутку называют *полом байтов*. Он важен, поскольку остроконечные и тупоконечные машины (а равно те, что используют иной порядок) часто общаются друг с другом по протоколам TCP/IP. Поскольку такая информация, как адреса отправления и назначения, номера портов, длина датаграмм, размеры окон и т.д., представляется в виде целых чисел, необходимо, чтобы обе стороны интерпретировали их одинаково.

Чтобы обеспечить взаимодействие компьютеров с разными архитектурами, все целочисленные величины, относящиеся к протоколам, передаются в *сетевом порядке байтов*, который по определению является тупоконечным. По большей

части, обо всем заботятся сами протоколы, но сетевые адреса, номера портов, а иногда и другие данные, представленные в заголовках, вы задаете сами. И всякий раз необходимо преобразовывать их в сетевой порядок.

Для этого служат две функции, занимающиеся преобразованием из машинного порядка байт в сетевой и обратно. Представленные ниже объявления этих функций заимствованы из стандарта POSIX. В некоторых версиях UNIX эти объявления находятся не в файле `netinet/in.h`. Типы `uint32_t` и `uint16_t` приняты в POSIX соответственно для беззнаковых 32- и 16-разрядных целых. В некоторых реализациях эти типы могут отсутствовать. Тем не менее функции `htonl` и `ntohl` всегда принимают и возвращают беззнаковые 32-разрядные целые числа, будь то UNIX или Winsock. Точно так же функции `htons` и `ntohs` всегда принимают и возвращают беззнаковые 16-разрядные целые.

Примечание *Буквы «l» и «s» в конце имен функций означают long (длинное) и short (короткое). Это имело смысл, так как первоначально данные функции появились в системе 4.2BSD, разработанной для 32-разрядной машины, где длинное целое принимали равным 32 бит, а короткое – 16. С появлением 64-разрядных машин это уже не так важно, поэтому следует помнить, что l-функции работают с 32-разрядными числами, которые не обязательно представлены как long, а s-функции – с 16 разрядными числами, которые не обязательно представлены в виде short. Удобно считать, что l-функции предназначены для преобразования длинных полей в заголовках протокола, а s-функции – коротких полей.*

```
#include <netinet/in.h>  /* UNIX */
#include <winsock2.h> /* Winsock */

uint32_t htonl( uint32_t host32 );

uint16_t htons( uint16_t host16 );
```

Обе функции возвращают целое число в сетевом порядке.

```
uint32_t ntohl( uint32_t network32 );

uint16_t ntohs( uint16_t network16 );
```

Обе функции возвращают целое число в машинном порядке.

Функции `htonl` и `htons` преобразуют целое число из машинного порядка байт в сетевой, тогда как функции `ntohl` и `ntohs` выполняют обратное преобразование. Заметим, что на «тупоконечных» машинах эти функции ничего не делают и обычно определяются в виде макросов:

```
#define htonl(x)   (x)
```

На «остроконечных» машинах (и для иных архитектур) реализация функций зависит от системы. Не надо задумываться, на какой машине вы работаете, поскольку эти функции всегда делают то, что нужно.

Применение этих функций *обязательно* только для полей, используемых протоколами. Пользовательские данные для протоколов IP, UDP и TCP выглядят как множество неструктурированных байтов, так что неважно, записаны целые числа в сетевом или машинном порядке. Тем не менее функции `ntoh*` и `hton*` стоит применять при передаче любых данных, поскольку тем самым вы обеспечиваете возможность совместной работы машин с разной архитектурой. Даже если сначала предполагается, что приложение будет работать только на одной платформе, обязательно настанет день, когда его придется переносить на другую платформу. Тогда дополнительные усилия окупятся с лихвой.

Примечание *В общем случае проблема преобразования данных между машинами с разными архитектурами сложна. Многие программисты решают ее, преобразуя все числа в код ASCII (или, возможно, в код EBCDIC для больших машин фирмы IBM). Другой подход связан с использованием компоненты XDR (External Data Representation – внешнее представление данных), входящей в состав подсистемы вызова удаленных процедур (RPC – remote procedure call), разработанной фирмой Sun. Компонента XDR определена в RFC 1832 [Srinivasan 1995] и представляет собой набор правил для кодирования данных различных типов, а также язык, описывающий способ кодирования. Хотя предполагалось, что XDR будет применяться как часть RPC, можно пользоваться этим механизмом в ваших программах. В книге [Stevens 1999] обсуждается XDR и его применение без RPC.*

И, наконец, следует помнить, что функции разрешения имен, такие как `gethostbyname` и `getservbyname` (совет 29), возвращают значения, представленные в сетевом порядке. Поэтому следующий неправильный код

```
struct servent *sp;
struct sockaddr_in *sap;
sp = getservbyname( name, protocol );
sap->sin_port = htons( sp->s_port );
```

приведет к ошибке, если исполняется не на «тупоконечной» машине.

Резюме

В этом разделе рассказывалось, что в TCP/IP применяется стандартное представление в сетевом порядке байт для целых чисел, входящих в заголовки протоколов. Здесь также приведены функции `htonl`, `htons`, `ntohl` и `ntohs`, которые преобразуют целые из машинного порядка байт в сетевой и обратно. Кроме того, было отмечено, что в общем случае для преобразования форматов данных между машинами полезно средство XDR.

Совет 29. Не «зашивайте» IP-адреса и номера портов в код

У программы есть только два способа получить IP-адрес или номер порта:

- из аргументов в командной строке или, если программа имеет графический интерфейс пользователя, с помощью диалогового окна либо аналогичного механизма;
- с помощью функции разрешения имен, например `gethostbyname` или `getservbyname`.

Примечание *Строго говоря, `getservbyname` – это не функция разрешения имени (то есть она не входит в состав DNS-клиента, который отображает имена на IP-адреса и наоборот). Но она рассмотрена вместе с остальными, поскольку выполняет похожие действия.*

Никогда не следует «зашивать» эти параметры в текст программы или помещать их в собственный (не системный) конфигурационный файл. И в UNIX, и в Windows есть стандартные способы получения этой информации, ими и надо пользоваться.

Теперь IP-адреса все чаще выделяются динамически с помощью протокола DHCP (dynamic host configuration protocol – протокол динамической конфигурации хоста). И это убедительная причина избегать их задания непосредственно в тексте программы. Некоторые считают, что из-за широкой распространенности DHCP и сложности адресов в протоколе IPv6 вообще не нужно передавать приложению числовые адреса, а следует ограничиться только символическими именами хостов, которые приложение должно преобразовать в IP-адреса, обратившись к функции `gethostbyname` или родственным ей. Даже если протокол DHCP не используется, управлять сетью будет намного проще, если не «зашивать» эту информацию в код и не помещать ее в нестандартные места. Например, если адрес сети изменяется, то все приложения с «зашитыми» адресами просто перестанут работать.

Всегда возникает искушение встроить адрес или номер порта непосредственно в текст программы, написанной «на скорую руку», и не возиться с функциями типа `getXbyY`. К сожалению, такие программы начинают жить своей жизнью, а иногда даже становятся коммерческими продуктами. Одно из преимуществ каркасов и библиотечных функций на их основе (совет 4) состоит в том, что код уже написан, так что нет необходимости «срезать углы».

Рассмотрим некоторые функции разрешения имен и порядок их применения. Вы уже не раз встречались с функцией `gethostbyname`:

```
#include <netdb.h>    /* UNIX */
#include <winsock2.h> /* Winsock */

struct hostent *gethostbyname( const char *name );
```

Возвращаемое значение: указатель на структуру `hostent` в случае успеха, NULL и код ошибки в переменной `h_errno` – в случае неудачи.

Функции gethostbyname передается имя хоста, а она возвращает указатель на структуру hostent следующего вида:

```
struct hostent {
   char *h_name;              /* Официальное имя хоста. */
   char **h_aliases;          /* Список синонимов. */
   int h_addrtype;            /* Тип адреса хоста. */
   int h_length;              /* Длина адреса. */
   char **h_addr_list;        /* Список адресов, полученных от DNS. */
#define h_addr h_addr_list[0]; /* Первый адрес. */
};
```

Поле h_name указывает на «официальное» имя хоста, а поле h_aliases – на список синонимов имени. Поле h_addrtype содержит либо AF_INET, либо AF_INET6 в зависимости от того, составлен ли адрес в соответствии с протоколом IPv4 или IPv6. Аналогично поле h_length равно 4 или 16 в зависимости от типа адреса. Все адреса типа h_addrtype возвращаются в списке, на который указывает поле h_addr_list. Макрос h_addr выступает в роли синонима первого (возможно, единственного) адреса в этом списке. Поскольку gethostbyname возвращает список адресов, приложение может попробовать каждый из них, пока не установит соединение с нужным хостом.

Работая с функцией gethostbyname нужно учитывать следующие моменты:

- если хост поддерживает оба протокола IPv4 и IPv6, то возвращается только один тип адреса. В UNIX тип возвращаемого адреса зависит от параметра RES_USE_INET6 системы разрешения имен, который можно явно задать, обратившись к функции res_init или установив переменную среду, а также с помощью опции в конфигурационном файле DNS. В соответствии с Winsock, всегда возвращается адрес IPv4;
- структура hostent находится в статической памяти. Это означает, что функция gethostbyname не рентабельна;
- указатели, хранящиеся в статической структуре hostent, направлены на другую статическую или динамически распределенную память, поэтому при желании скопировать структуру необходимо выполнять *глубокое* копирование. Это означает, что помимо памяти для самой структуры hostent необходимо выделить память для каждой области, на которую указывают поля структуры, а затем скопировать в нее данные;
- как говорилось в совете 28, адреса, хранящиеся в списке, на который указывает поле h_addr_list, уже приведены к сетевому порядку байтов, так что применять к ним функцию htonl не надо.

Вы можете также выполнить обратную операцию – отобразить адреса хостов на их имена. Для этого служит функция gethostbyaddr.

```
#include <netdb.h>    /* UNIX. */
#include <winsock2.h> /* Winsock. */

struct hostent *gethostbyaddr( const char *addr, int len, int type );
```

Возвращаемое значение: указатель на структуру hostent в случае успеха, NULL и код ошибки в переменной h_errno – в случае неудачи.

Несмотря на то, что параметр *addr* имеет тип char*, он указывает на структуру in_addr (или in6_addr в случае IPv6). Длина этой структуры задается параметром *len*, а ее тип (AF_INET или AF_INET6) – параметром *type*. Предыдущие замечания относительно функции gethostbyname касаются и gethostbyaddr.

Для хостов, поддерживающих протокол IPv6, функции gethostbyname недостаточно, так как нельзя задать тип возвращаемого адреса. Для поддержки IPv6 (и других адресных семейств) введена общая функция gethostbyname2, допускающая получение адресов указанного типа.

```
#include <netdb.h>    /* UNIX */

struct hostent *gethostbyname2( const char *name, int af );
```

Возвращаемое значение: указатель на структуру hostent в случае успеха, NULL и код ошибки в переменной h_errno – в случае неудачи.

Параметр *af* – это адресное семейство. Интерес представляют только возможные значения AF_INET или AF_INET6. Спецификация Winsock не определяет функцию gethostbyname2, а использует вместо нее функционально более богатый (и сложный) интерфейс WSALookupServiceNext.

Примечание *Взаимодействие протоколов IPv4 и IPv6 – это в значительной мере вопрос обработки двух разных типов адресов. И функция gethostbyname2 предлагает один из способов решения этой проблемы. Эта тема подробно обсуждается в книге [Stevens 1998], где также приведена реализация описанной в стандарте POSIX функции getaddrinfo. Эта функция дает удобный, не зависящий от протокола способ работы с обоими типами адресов. С помощью getaddrinfo можно написать приложение, которое будет одинаково работать и с IPv4, и с IPv6.*

Раз системе (или службе DNS) разрешено преобразовывать имена хостов в IP-адреса, почему бы ни сделать то же и для номеров портов? В совете 18 рассматривался один способ решения этой задачи, теперь остановимся на другом. Так же, как gethostbyname и gethostbyaddr выполняют преобразование имени хоста в адрес и обратно, функции getservbyname и getservbyport преобразуют символическое имя сервиса в номер порта и наоборот. Например, сервис времени дня daytime прослушивает порт 13 в ожидании TCP-соединений или UDP-датаграмм. Можно обратиться к нему, например, с помощью программы telnet:

```
telnet bsd 13
```

Однако необходимо учитывать, что номер порта указанного сервиса равен 13. К счастью, telnet понимает и символические имена портов:

```
telnet bsd daytime
```

Telnet выполняет отображение символических имен на номера портов, вызывая функцию getservbyname; вы сделаете то же самое. В листинге 2.3 вы

увидите, что в предложенном каркасе этот вызов уже есть. Функция set_address сначала оперирует параметром port как представленным в коде ASCII целым числом, то есть пытается преобразовать его в двоичную форму. Если это не получается, то вызывается функция getservbyname, которая ищет в базе данных символическое имя порта и возвращает соответствующее ему числовое значение.

Прототип функции getservbyname похож на gethostbyname:

```
#include <netdb.h>    /* UNIX */
#include <winsock2.h> /* Winsock */

struct servent *getservbyname( const char *name, const char *proto );
```

Возвращаемое значение: указатель на структуру servent в случае успеха, NULL – в случае неудачи.

Параметр *name* – это символическое имя сервиса, например «daytime». Если параметр *proto* не равен NULL, то возвращается сервис, соответствующий заданным имени и типу протокола, в противном случае – первый найденный сервис с именем name. Структура servent содержит информацию о найденном сервисе:

```
struct servent {
   char *s_name;       /* Официальное имя сервиса. */
   char **s_aliases;   /* Список синонимов. */
   int s_port;         /* Номер порта. */
   char *s_proto;      /* Используемый протокол. */
};
```

Поля s_name и s_aliases содержат указатели на официальное имя сервиса и его синонимы. Номер порта сервиса находится в поле s_port. Как обычно, этот номер уже представлен в сетевом порядке байтов. Протокол (TCP или UDP), используемый сервисом, описывается строкой в поле s_proto.

Вы можете также выполнить обратную операцию – найти имя сервиса по номеру порта. Для этого служит функция getservbyport:

```
#include <netdb.h>    /* UNIX. */
#include <winsock2.h> /* Winsock. */

struct servent *getservbyport( int port, const char *proto );
```

Возвращаемое значение: указатель на структуру servent в случае успеха, NULL – в случае неудачи

Передаваемый в параметре *port* номер порта должен быть записан в сетевом порядке. Параметр *proto* имеет тот же смысл, что и раньше.

С точки зрения программиста, данный каркас и библиотечные функции решают задачи преобразования имен хостов и сервисов. Они сами вызывают нужные функции, а как это делается, не должно вас волновать. Однако нужно знать, как ввести в систему необходимую информацию.

Обычно это делается с помощью одного из трех способов:

- DNS;
- сетевой информационной системы (NIS) или NIS+;
- файлов `hosts` и `services`.

DNS (Domain Name System – служба доменных имен) – это распределенная база данных для преобразования имен хостов в адреса.

Примечание *DNS используется также для маршрутизации электронной почты. Когда посылается письмо на адрес* `jsmith@somecompany.com`*, с помощью DNS ищется обработчик (или обработчики) почты для компании* `somecompany.com`*. Подробнее это объясняется в книге [Albitz and Lin 1998].*

Ответственность за хранение данных распределяется между зонами (грубо говоря, они соответствуют адресным доменам) и подзонами. Например, bigcompany.com может представлять собой одну зону, разбитую на несколько подзон, соответствующих отделам или региональным отделениям. В каждой зоне и подзоне работает один или несколько DNS-серверов, на которых хранится вся информация о хостах в этой зоне или подзоне. Другие DNS-серверы могут запросить информацию у данных серверов для разрешения имен хостов, принадлежащих компании BigCompany.

Примечание *Система DNS – это хороший пример UDP-приложения. Как правило, обмен с DNS-сервером происходит короткими транзакциями. Клиент (обычно одна из функций разрешения имен) посылает UDP-датаграмму, содержащую запрос к DNS-серверу. Если в течение некоторого времени ответ не получен, то пробуется другой сервер, если таковой известен. В противном случае повторно посылается запрос первому серверу, но с увеличенным тайм-аутом.*

На сегодняшний день подавляющее большинство преобразований между именами хостов и IP-адресами производится с помощью службы DNS. Даже сети, не имеющие выхода вовне, часто пользуются DNS, так как это упрощает администрирование. При добавлении в сеть нового хоста или изменении адреса существующего нужно обновить только базу данных DNS, а не файлы `hosts` на каждой машине.

Система NIS и последовавшая за ней NIS+ предназначены для ведения централизованной базы данных о различных аспектах системы. Помимо имен хостов и IP-адресов, NIS может управлять именами сервисов, паролями, группами и другими данными, которые следует распространять по всей сети. Стандартные функции разрешения имен (о них говорилось выше) могут опрашивать и базы данных NIS. В некоторых системах NIS-сервер при получении запроса на разрешение имени хоста, о котором у него нет информации, автоматически посылает запрос DNS-серверу. В других системах этим занимается функция разрешения имен.

Преимущество системы NIS в том, что она централизует хранение всех распространяемых по сети данных, упрощая тем самым администрирование больших

сетей. Некоторые эксперты не рекомендуют NIS, так как имеется потенциальная угроза компрометации паролей. В системе NIS+ эта угроза снята, но все равно многие опасаются пользоваться ей. NIS обсуждается в работе [Brown 1994].

Последнее и самое неудобное из стандартных мест размещения информации об именах и IP-адресах хостов – это файл `hosts`, обычно находящийся в каталоге `/etc` на каждой машине. В этом файле хранятся имена, синонимы и IP-адреса хостов в сети. Стандартные функции разрешения имен просматривают также и этот файл. Обычно при конфигурации системы можно указать, когда следует просматривать файл `hosts` – до или после обращения к службе DNS.

Другой файл – обычно `/etc/services` – содержит информацию о соответствии имен и портов сервисов. Если NIS не используется, то, как правило, на каждой машине имеется собственная копия этого файла. Поскольку он изменяется редко, с его администрированием не возникает таких проблем, как с файлом `hosts`. В совете 17 было сказано о формате файла `services`.

Основной недостаток файла `hosts` – это очевидное неудобство его сопровождения. Если в сети более десятка хостов, то проблема быстро становится почти неразрешимой. В результате многие эксперты рекомендуют полностью отказаться от такого метода. Например, в книге [Lehey 1996] советуется следующее: «Есть только одна причина не пользоваться службой DNS – если ваш компьютер не подсоединен к сети».

Резюме

В этом разделе рекомендовано не «зашивать» адреса и номера портов в программу. Также рассмотрено несколько стандартных схем получения этой информации и обсуждены их достоинства и недостатки.

Совет 30. Разберитесь, что такое подсоединенный UDP-сокет

Здесь рассказывается об использовании вызова `connect` применительно к протоколу UDP. Из совета 1 вам известно, что UDP – это протокол, не требующий установления соединений. Он передает отдельные адресованные конкретному получателю датаграммы, поэтому кажется, что слово «connect» (соединить) тут неуместно. Следует, однако, напомнить, что в листинге 3.6 вы уже встречались с примером, где вызов `connect` использовался в запускаемом через `inetd` UDP-сервере, чтобы получить (эфемерный) порт для этого сервера. Только так `inetd` мог продолжать прослушивать датаграммы, поступающие в исходный хорошо известный порт.

Прежде чем обсуждать, зачем нужен вызов `connect` для UDP-сокета, вы должны четко представлять себе, что собственно означает «соединение» в этом контексте. При использовании TCP вызов `connect` инициирует обмен информацией о состоянии между сторонами с помощью процедуры трехстороннего квитирования (рис. 3.14). Частью информации о состоянии является адрес и порт каждой стороны, поэтому можно считать, что одна из функций вызова `connect` в протоколе TCP – это привязка адреса и порта удаленного хоста к локальному сокету.

Хотя полезность вызова connect в протоколе UDP может показаться сомнительной, но вы увидите, что, помимо некоторого повышения производительности, он позволяет выполнить такие действия, которые без него были бы невозможны. Рассмотрим причины использования соединенного сокета UDP сначала с точки зрения отправителя, а потом – получателя.

Прежде всего, от подсоединенного UDP-сокета вы получаете возможность использования вызова send или write (в UNIX) вместо sendto.

Примечание *Для подсоединенного UDP-сокета можно использовать и вызов* sendto, *но в качестве указателя на адрес получателя надо задавать* NULL, *а в качестве его длины – нуль. Возможен, конечно, и вызов* sendmsg, *но и в этом случае поле* msg_name *в структуре* msghdr *должно содержать* NULL, *а поле* msg_namelen *– нуль.*

Само по себе это, конечно, немного, но все же вызов connect действительно дает заметный выигрыш в производительности.

В реализации BSD sendto – это частный случай connect. Когда датаграмма посылается с помощью sendto, ядро временно соединяет сокет, отправляет датаграмму, после чего отсоединяет сокет. Изучая систему 4.3BSD и тесно связанную с ней SunOS 4.1.1, Партридж и Пинк [Partridge and Pink 1993] заметили, что такой способ соединения и разъединения занимает почти треть времени, уходящего на передачу датаграммы. Если не считать усовершенствования кода, который служит для поиска *управляющего блока протокола* (PCB – protocol control block) и ассоциирован с сокетом, исследованный этими авторами код почти без изменений вошел в систему 4.4BSD и основанные на ней, например FreeBSD. В частности, эти стеки по-прежнему выполняют временное соединение и разъединение. Таким образом, если вы собираетесь посылать последовательность UDP-датаграмм одному и тому же серверу, то эффективность можно повысить, предварительно вызвав connect.

Этот выигрыш в производительности характерен только для некоторых реализаций. А основная причина, по которой отправитель UDP-датаграмм подсоединяет сокет, – это желание получать уведомления об *асинхронных событиях*. Представим, что надо послать UDP-датаграмму, но никакой процесс на другой стороне не прослушивает порт назначения. Протокол UDP на другом конце вернет ICMP-сообщение о недоступности порта, информируя тем самым ваш стек TCP/IP, но если сокет не подсоединен, то приложение не получит уведомления. Когда вы вызываете sendto, в начало сообщения добавляется заголовок, после чего оно передается уровню IP, где инкапсулируется в IP-датаграмму и помещается в выходную очередь интерфейса. Как только датаграмма внесена в очередь (или отослана, если очередь пуста), sendto возвращает управление приложению с кодом нормального завершения. Иногда через некоторое время (отсюда и термин *асинхронный*) приходит ICMP-сообщение от хоста на другом конце. Хотя в нем есть копия UDP-заголовка, у вашего стека нет информации о том, какое приложение посылало датаграмму (вспомните совет 1, где говорилось, что из-за отсутствия установленного соединения система сразу забывает об отправленных датаграммах).

Если же сокет подсоединен, то этот факт отмечается в управляющем блоке протокола, связанном с сокетом, и стек TCP/IP может сопоставить полученную копию UDP-заголовка с тем, что хранится в PCB, чтобы определить, в какой сокет направить ICMP-сообщение.

Можно проиллюстрировать данную ситуацию с помощью вашей программы udpclient (листинг 3.5) из совета 17 – следует отправить датаграмму в порт, который не прослушивает ни один процесс:

```
bsd: $ udpclient bsd 9000
Hello, World!
^C                Клиент "зависает" и прерывается вручную.
bsd: $
```

Теперь модифицируем клиент, добавив такие строки

```
if ( connect( s, ( struct sockaddr * )&peer, sizeof( peer ) ) )
    error( 1, errno, "ошибка вызова connect" );
```

сразу после вызова функции udp_client. Если назвать эту программу udpconn1 и запустить ее, то вы получите следующее:

```
bsd: $ udpconn1 bsd 9000
Hello, World!
updconn1: ошибка вызова sendto: Socket is already connected (56)
bsd: $
```

Ошибка произошла из-за того, что вы вызвали sendto для подсоединенного сокета. При этом sendto потребовал от UDP временно подсоединить сокет. Но UDP определил, что сокет уже подсоединен и вернул код ошибки EISCONN.

Чтобы исправить ошибку, нужно заменить обращение к sendto на

```
rc = send( s, buf, strlen( buf ), 0 );
```

Назовем новую программу udpconn2. После ее запуска получится такой результат:

```
bsd: $ udpconn1 bsd 9000
Hello, World!
updconn2: ошибка вызова recvfrom: Connection refused (61)
bsd: $
```

На этот раз ошибку ECONNREFUSED вернул вызов recvfrom. Эта ошибка – результат получения приложением ICMP-сообщения о недоступности порта.

Обычно у получателя нет причин соединяться с отправителем (если, конечно, ему самому не нужно стать отправителем). Однако иногда такая ситуация может быть полезна. Вспомним аналогию с телефонным разговором и почтой (совет 1). TCP-соединение похоже на частный телефонный разговор – в нем только два участника. Поскольку в протоколе TCP устанавливается соединение, каждая сторона знает о своем партнере и может быть уверена, что всякий полученный байт действительно послал партнер.

С другой стороны, приложение, получающее UDP-датаграммы, можно сравнить с почтовым ящиком. Как любой человек может отправить письмо по данному

адресу, так и любое приложение или хост может послать приложению-получателю датаграмму, если известны адрес и номер порта.

Иногда нужно получать датаграммы только от одного приложения. Получающее приложение добивается этого, соединившись со своим партнером. Чтобы увидеть, как это работает, напишем UDP-сервер эхо-контроля, который соединяется с первым клиентом, отправившим датаграмму (листинг 3.35).

Листинг 3.35. UDP-сервер эхо-контроля, выполняющий соединение

udpconnserv.c

```
 1 #include "etcp.h"
 2 int main( int argc, char **argv )
 3 {
 4    struct sockaddr_in peer;
 5    SOCKET s;
 6    int rc;
 7    int len;
 8    char buf[ 120 ];

 9    INIT();
10    s = udp_server( NULL, argv[ 1 ] );
11    len = sizeof( peer );
12    rc = recvfrom( s, buf, sizeof( buf ),
13       0, ( struct sockaddr * )&peer, &len );
14    if ( rc < 0 )
15       error( 1, errno, "ошибка вызова recvfrom" );
16    if ( connect( s, ( struct sockaddr * )&peer, len ) )
17       error( 1, errno, "ошибка вызова connect" );

18    while ( strncmp( buf, "done", 4 ) != 0 )
19    {
20       if ( send( s, buf, rc, 0 ) < 0 )
21          error( 1, errno, "ошибка вызова send" );
22       rc = recv( s, buf, sizeof( buf ), 0 );
23       if ( rc < 0 )
24          error( 1, errno, "ошибка вызова recv" );
25    }
26    EXIT( 0 );
27 }
```

udpconnserv.c

9-15 Выполняем стандартную инициализацию UDP и получаем первую датаграмму, сохраняя при этом адрес и порт отправителя в переменной `peer`.

16-17 Соединяемся с отправителем.

18-25 В цикле отсылаем копии полученных датаграмм, пока не придет датаграмма, содержащая единственное слово «done».

Для экспериментов с сервером `udpconnserv` можно воспользоваться клиентом `udpconn2`. Сначала запускается сервер для прослушивания порта 9000 в ожидании датаграмм:

```
udpconnserv 9000
```

а затем запускаются две копии udpconn2, каждая в своем окне.

```
bsd: $ udpconn2 bsd 9000
one
one

three
three
done
^C
bsd: $
```

```
bsd: $ udpconn2 bsd 9000
two
udpconn2: ошибка вызова recvfrom:
          Connection refused (61)
bsd: $
```

Когда в первом окне вы набираете one, сервер udpconnserv возвращает копию датаграммы. Затем во втором окне вводите two, но recvfrom возвращает код ошибки ECONNREFUSED. Это происходит потому, что UDP вернул ICMP-сообщение о недоступности порта, так как ваш сервер уже соединился с первым экземпляром udpconn2 и не принимает датаграммы с других адресов.

Примечание *Адреса отправителя у обоих экземпляров udpconn2, конечно, одинаковы, но эфемерные порты, выбранные стеком TCP/IP, различны. В первом окне вы набираете* **three**, *дабы убедиться, что udpconnserv все еще функционирует, а затем –* **done**, *чтобы остановить сервер. В конце прерываем вручную первый экземпляр udpconn2.*

Как видите, udpconnserv не только отказывается принимать датаграммы от другого отправителя, но также информирует приложение об этом факте, посылая ICMP-сообщение. Разумеется, чтобы получить это сообщение, клиент также должен подсоединиться к серверу. Если бы вы прогнали этот тест с помощью первоначальной версии клиента udpclient вместо udpconn2, то второй экземпляр клиента просто «завис» после ввода слова «done».

Резюме

В этом разделе рассмотрено использование вызова connect в протоколе UDP. Хотя на первый взгляд может показаться, что для протокола без установления соединения это не имеет смысла, но, как вы видели, такое действие, во-первых, повышает производительность, а во-вторых, оно необходимо при желании получать некоторые сообщения об ошибках при отправке UDP-датаграмм. Здесь также описано, как использовать connect для приема датаграмм только от одного хоста.

Совет 31. Помните, что С – не единственный язык программирования

До сих пор все примеры в этой книге были написаны на языке С, но, конечно, это не единственно возможный выбор. Многие предпочитают писать на С++, Java или даже Pascal. В этом разделе будет рассказано об использовании языков сценариев для сетевого программирования и приведено несколько примеров на языке Perl.

Вы уже встречались с несколькими примерами небольших программ, написанных специально для тестирования более сложных приложений. Например, в совете 30 использованы простые и похожие программы udpclient, udpconn1 и udpconn2 для проверки поведения подсоединенного UDP-сокета. В таких случаях имеет смысл воспользоваться каким-либо языком сценариев. Сценарии проще разрабатывать и модифицировать хотя бы потому, что их не надо компилировать и компоновать со специальной библиотекой, а также создавать файлы сборки проекта (Makefile) – достаточно написать сценарий и сразу же запустить его.

В листинге 3.36 приведен текст минимального Perl-сценария, реализующего функциональность программы udpclient.

Хотя я не собираюсь писать руководство по языку Perl, но этот пример стоит изучить подробнее.

Примечание *Глава 6 стандартного учебника по Perl [Wall et al. 1996] посвящена имеющимся в этом языке средствам межпроцессного взаимодействия и сетевого программирования. Дополнительную информацию о языке Perl можно найти на сайте http://www.perl.com.*

Листинг 3.36. Версия программы udpclient на языке Perl

```
                                                                  pudpclient
 1 #! /usr/bin/perl5
 2 use Socket;
 3 $host = shift || "localhost";
 4 $port = shift || "echo";
 5 $port = getservbyname( $port, "udp" ) if $port =~ /\D/;
 6 $peer = sockaddr_in( $port, inet_aton( $host ) );
 7 socket( S, PF_INET, SOCK_DGRAM, 0 ) || die "ошибка вызова socket $!";
 8 while ( $line = <STDIN> )
 9 {
10 defined( send( S, $line, 0, $peer ) ) || die "ошибка вызова send $!";
11 defined( recv( S, $line, 120, 0 ) ) || die "ошибка вызова recv $!";
12 print $line;
13 }
                                                                  pudpclient
```

Инициализация

2 В этой строке Perl делает доступными сценарию определения некоторых констант (например, PF_INET).

Получение параметров командной строки

3-4 Из командной строки читаем имя хоста и номер порта. Обратите внимание, что этот сценарий делает больше, чем программа на языке C, так как по умолчанию он присваивает хосту имя localhost, а порту – echo, если один или оба параметра не заданы явно.

Заполнение структуры sockaddr_in и получение сокета

5-6 Этот код выполняет те же действия, что и функция set_address в листинге 2.3 в совете 4. Обратите внимание на простоту кода. В этих двух

строчках IP-адрес хоста принимается как числовой и его имя символическое, а равно числовое или символическое имя сервиса.

7 Получаем UDP-сокет.

Основной цикл

8-13 Так же, как в `udpclient`, читаем строки из стандартного ввода, отправляем их удаленному хосту, читаем от него ответ и записываем его на стандартный вывод.

Хотя знакомые сетевые функции иногда принимают несколько иные аргументы и могут возвращать результат непривычным образом, но, в общем, программа в листинге 3.36 кажется знакомой и понятной. Тот, кто знаком с основами сетевого программирования и хотя бы чуть-чуть разбирается в Perl, может добиться высокой производительности.

Для сравнения в листинге 3.37 представлен TCP-сервер эхо-контроля. Вы можете соединиться с этим сервером с помощью программы telnet или любого другого TCP-приложения, способного вести себя как клиент эхо-сервера.

Здесь также видна знакомая последовательность обращений к API сокетов и, даже не зная языка Perl, можно проследить за ходом выполнения программы. Следует отметить две особенности, присущие Perl:

- вызов `accept` на строке 11 возвращает `TRUE`, если все хорошо, а новый сокет возвращается во втором параметре (*S1*). В результате естественно выглядит цикл `for`, в котором принимаются соединения;
- поскольку `recv` возвращает адрес отправителя (или специальное значение `undef`), а не число прочитанных байт, получая длину строки `$line` (строка 16), следует явно проверять, не пришел ли признак конца файла. Оператор `last` выполняет те же действия, что `break` в языке C.

Листинг 3.37. Версия эхо-сервера на языке Perl

pechos

```
 1 #! /usr/bin/perl5
 2 use Socket;
 3 $port = shift;
 4 $port = getservbyname( $port, 'tcp' ) if $port =~ /\D/;
 5 die "Invalid port" unless $port;
 6 socket( S, PF_INET, SOCK_STREAM, 0 ) || die "socket: $!";
 7 setsockopt( S, SOL_SOCKET, SO_REUSEADDR, pack( 'l', 1 ) ) ||
 8 die "setsockopt: $!";
 9 bind( S, sockaddr_in( $port, INADDR_ANY ) ) || die "bind: $!";
10 listen( S, SOMAXCONN );
11 for( ; accept( S1, S ); close( S1 ) )
12 {
13    while ( TRUE )
14    {
15       defined( recv( S1, $line, 120, 0 ) ) || die "recv: $!";
16       last if length( $line ) == 0;
```

```
        defined( send( S1, $line, 0 ) ) || die "send: $!";
    }
}
```

pechos

Как видно из этих двух примеров, языки сценариев вообще и Perl в частности – это отличный инструмент для написания небольших тестовых программ, создания прототипов более крупных систем и утилит. Perl и другие языки сценариев активно применяются при разработке Web-серверов и специализированных Web-клиентов. Примеры рассматриваются в книгах [Castro 1998] и [Patchett and Wright 1998].

Помимо простоты и скорости разработки прототипа, есть и другие причины для использования языков сценариев. Одна из них – наличие в таких языках специальных возможностей. Например, Perl обладает прекрасными средствами для манипулирования данными и работы с регулярными выражениями. Поэтому во многих случаях Perl оказывается удобнее таких традиционных языков, как C.

Предположим, что каждое утро вам надо проверять, не появились ли в конференции comp.protocols.tcp-ip новые сообщения о протоколах TCP и UDP. В листинге 3.38 приведен каркас Perl-сценария для автоматизации решения этой задачи. В таком виде сценарий не очень полезен, так как он показывает все сообщения от сервера новостей, даже старые; отбор сообщений осуществляется довольно грубо. Можно было бы без труда модифицировать сценарий, ужесточив критерий отбора, но лучше оставить его таким, как есть, чтобы не запутаться в деталях языка Perl. Подробнее протокол передачи сетевых новостей (NNTP) рассматривается в RFC 977 [Kantor and Lapsley 1986].

Листинг 3.38. Perl-сценарий для формирования дайджеста из сетевых конференций

tcpnews

```
#! /usr/bin/perl5
use Socket;
$host = inet_aton( 'nntp.ix.netcom.com') || die "хост: $!";
$port = getservbyname( 'nntp', 'tcp' ) || die "некорректный порт";
socket( S, PF_INET, SOCK_STREAM, 0 ) || die "socket: $!";
connect( S, sockaddr_in( $port, $host ) ) || die "connect: $!";
select( S );
$| = 1;
select( STDOUT );
print S "group comp.protocols.tcp-ip\r\n";
while ( $line = <S> )
{
    last if $line =~ /^211/;
}
($rc, $total, $start, $end ) = split( /\s/, $line );
print S "xover $start-$end\nquit\r\n";
while ( $line = <S> )
{
    ( $no, $sub, $auth, $date ) = split( /\t/, $line );
```

```
20    print "$no, $sub, $date\n" if $sub =~ /TCP|UDP/;
21 }
22 close( S );
```

tcpnews

Инициализация и соединение с сервером новостей

2-6 Это написанный на Perl аналог логики инициализации стандартного TCP-клиента.

Установить режим небуферизованного ввода/вывода

7-9 В Perl функция `print` вызывает стандартную библиотеку ввода/вывода, а та, как упоминалось в совете 17, буферизует вывод в сокет. Эти три строки отключают буферизацию. Хотя по виду оператор `select` напоминает системный вызов `select`, который рассматривался ранее, в действительности он просто указывает, какой файловый дескриптор будет использоваться по умолчанию. Выбрав дескриптор, вы можете отменить буферизацию вывода в сокет `S`, задав ненулевое значение специальной переменной `$|`, используемой в Perl.

Примечание *Строго говоря, это не совсем так. Эти действия приводят к тому, что после каждого вызова `write` или `print` для данного дескриптора автоматически выполняется функция `fflush`. Но результат оказывается таким же, как если бы вывод в сокет был не буферизован.*

В строке 9 `stdout` восстанавливается как дескриптор по умолчанию.

Выбрать группу comp.protocols.tcp-ip

10-14 Посылаем серверу новостей команду `group`, которая означает, что текущей группой следует сделать `comp.protocols.tcp-ip`. Сервер отвечает строкой вида

```
211 total_articles first_article# last_article# group_namespace
```

В строке 13 вы ищете именно такой ответ, отбрасывая все строки, которые начинаются не с кода ответа 211. Обратите внимание, что оператор `<...>` сам разбивает на строки входной поток, поступающий от TCP.

15-16 Обнаружив ответ на команду `group`, нужно послать серверу строки

```
xover first_article#-last_article#
quit
```

Команда `xover` запрашивает сервер, заголовки всех статей с номерами из заданного диапазона. Заголовок содержит список данных, разделенных символами табуляции: номер статьи, тема, автор, дата и время, идентификатор сообщения, идентификаторы сообщений для статей, на которую ссылается данная, число байтов и число строк. Команда `quit` приказывает серверу разорвать соединение, так как запросов больше не будет.

Отбор заголовков статей

17-20 Читаем каждый заголовок, выделяем из него интересующие нас поля и оставляем только те заголовки, для которых в теме присутствует строка «TCP» или «UDP».

Запуск `tcpnews` дает следующий результат:

```
bsd: $ tcpnews
74179, Re: UDP multicast, Thu, 22 Jul 1999 21:06:47 GMT
74181, Re: UDP multicast, Thu, 22 Jul 1999 21:10:45 -0500
74187, Re: UDP multicast, Thu, 22 Jul 1999 23:23:00 +0200
74202, Re: NT 4.0 Server and TCP/IP, Fri, 23 Jul 1999 11:56:07 GMT
74227, New Seiko TCP/IP Chip, Thu, 22 Jul 1999 08:39:09 -0500
74267, WATTCP problems, Mon, 26 Jul 1999 13:18:14 -0500
74277, Re: New Seiko TCP/IP Chip, Thu, 26 Jul 1999 23:33:42 GMT
74305, TCP Petri Net model, Wed, 28 Jul 1999 02:27:20 +0200
bsd: $
```

Помимо языка Perl, есть и другие языки сценариев, пригодные для сетевого программирования, например:

- TCL/Expect;
- Python;
- JavaScript;
- Visual Basic (для Windows).

Их можно использовать для автоматизации решения простых сетевых задач, построения прототипов и быстрого создания удобных утилит или тестовых примеров. Как вы видели, языки сценариев часто проще применять, чем традиционные компилируемые языки программирования, поскольку интерпретатор берет на себя многие технические детали (конечно, расплачиваясь эффективностью). Усилия, потраченные на овладение хотя бы одним из таких языков, окупятся ростом производительности труда.

Резюме

В этом разделе говорилось об использовании языков сценариев в сетевом программировании. Нередко их применение имеет смысл при написании небольших утилит и тестовых программ.

Совет 32. Определите, на что влияют размеры буферов

Здесь приводятся некоторые эвристические правила для задания размеров буферов приема и передачи в TCP. В совете 7 обсуждалось, как задавать эти размеры с помощью функции `setsockopt`. Теперь вы узнаете, какие значения следует устанавливать.

Необходимо сразу отметить, что выбор правильного размера буфера зависит от приложения. Для интерактивных приложений типа telnet желательно устанавливать небольшой буфер. Этому есть две причины:

- обычно клиент посылает небольшой блок данных серверу и ждет ответа. Поэтому выделять большой буфер для таких соединений – пустая трата системных ресурсов;
- при большом буфере реакция на действия пользователя происходит не сразу. Например, если пользователь выводит на экран большой файл и нажимает комбинацию клавиш прерывания (**Ctrl+C**), то вывод не прекратится, пока в буферах есть данные. Если буфер велик, то до реального прерывания просмотра может пройти заметное время.

Знать размер буфера необходимо для расчета максимальной производительности. Например, это относится к приложениям, которые передают большие объемы данных преимущественно в одном направлении. Далее будут рассматриваться именно такие приложения.

Как правило, для получения максимальной пропускной способности рекомендуется, чтобы размеры буферов приема и передачи были не меньше *произведения полосы пропускания на задержку*. Как вы увидите, это правильный, но не слишком полезный совет. Прежде чем объяснять причины, разберемся, что представляет собой это произведение и почему размер буферов «правильный».

Вы уже несколько раз встречались с периодом кругового обращения (RTT). Это время, которое требуется пакету на «путешествие» от одного хоста на другой и обратно. Оно и представляет собой множитель «задержки», поскольку определяет время между моментом отправки пакета и подтверждением его получателем. Обычно RTT измеряется в миллисекундах.

Другой множитель в произведении – *полоса пропускания* (bandwidth). Это количество данных, которое можно передать в единицу времени по данному физическому носителю.

Примечание *Технически это не совсем корректно, но этот термин уже давно используется.*

Полоса пропускания измеряется в битах в секунду. Например, для сети Ethernet полоса пропускания (чистая) равна 10 Мбит/с.

Произведение полосы пропускания на задержку BWD вычисляется по формуле:

```
BWD = bandwidth × RTT.
```

Если RTT выражается в секундах, то единица измерения BWD будет следующей:

$$BWD = \frac{\text{бит}}{\text{секунда}} \times \text{секунда} = \text{бит}.$$

Если представить коммуникационный канал как «трубу», то произведение полосы пропускания на задержку – это объем трубы в битах (рис. 3.15), то есть количество данных, которые могут находиться в сети в любой момент времени.

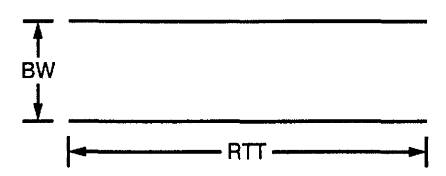


Рис. 3.15 Труба емкостью BWD бит

А теперь представим, как выглядит эта труба в установившемся режиме (после завершения алгоритма медленного старта) при массовой передаче данных, занимающей всю доступную полосу пропускания. Отправитель слева на рис. 3.16 заполнил трубу TCP-сегментами и должен ждать, пока сегмент n покинет сеть. Только после этого он сможет послать следующий сегмент. Поскольку в трубе находится столько же сегментов АСК, сколько и сегментов данных, при получении подтверждения на сегмент n – 8 отправитель может заключить, что сегмент n покинул сеть.

Это иллюстрирует феномен *самосинхронизации* (self-clocking property) TCP-соединения в установившемся режиме [Jacobson 1988]. Полученный сегмент АСК служит сигналом для отправки следующего сегмента данных.

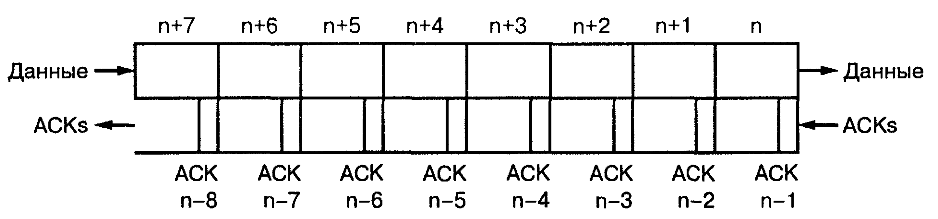


Рис. 3.16. Сеть в установившемся режиме

Примечание *Этот механизм часто называют АСК-таймером (ACK clock).*

Если необходимо, чтобы механизм самосинхронизации работал и поддерживал трубу постоянно заполненной, то окно передачи должно быть достаточно велико для обеспечения 16 неподтвержденных сегментов (от n – 8 до n + 7). Это означает, что буфер передачи на вашей стороне и буфер приема на стороне получателя должны иметь соответствующий размер для хранения 16 сегментов. В общем случае необходимо, чтобы в буфере помещалось столько сегментов, сколько находится в заполненной трубе. Значит, размер буфера должен быть не меньше произведения полосы пропускания на задержку.

Выше было отмечено, что это правило не особенно полезно. Причина в том, что обычно трудно узнать величину этого произведения. Предположим, что вы пишете приложение типа FTP. Насколько велики должны быть буферы приема и передачи? Во время написания программы неясно, какая сеть будет использоваться, а поэтому неизвестна и ее полоса пропускания. Но даже если это можно узнать во время выполнения, опросив сетевой интерфейс, то остается еще неизвестной величина задержки. В принципе, ее можно оценить с помощью какого-нибудь механизма типа `ping`, но, скорее всего, задержка будет варьироваться в течение существования соединения.

Примечание *Одно из возможных решений этой проблемы предложено в работе [Semke et al.]. Оно состоит в динамическом изменении размеров буферов. Авторы замечают, что размер окна перегрузки можно рассматривать как оценку произведения полосы пропускания на задержку. Подбирая размеры буферов в соответствии с текущим размером окна перегрузки (конечно, применяя подходящее демпфирование и ограничения, обеспечивающие справедливый режим для всех приложений), они сумели получить очень высокую производительность на одновременно установленных соединениях с разными величинами BWD. К сожалению, такое решение требует изменения в ядре операционной системы, так что прикладному программисту оно недоступно.*

Как правило, размер буферов назначают по величине, заданной по умолчанию или большей. Однако ни то, ни другое решение не оптимально. В первом случае может резко снизиться пропускная способность, во втором, как сказано в работе [Semke et al. 1998], – исчерпаны буферы, что приведет к сбою операционной системы.

В отсутствии априорных знаний о среде, в которой будет работать приложение, наверное, лучше всего использовать маленькие буферы для интерактивных приложений и буферы размером 32–64 Кб – для приложений, выполняющих массовую передачу данных. Однако не забывайте, что при работе в высокоскоростных сетях следует задавать намного больший размер буфера, чтобы использовать всю доступную полосу пропускания. В работе [Mahdavi 1997] приводятся некоторые рекомендации по оптимизации настройки стеков TCP/IP.

Есть одно правило, которое легко применять на практике, позволяющее повысить общую производительность во многих реализациях. В работе [Comer and Lin 1995] описывается эксперимент, в ходе которого два хоста были соединены сетью Ethernet в 10 Мбит и сетью ATM в 100 Мбит. Когда использовался размер буфера 16 Кб, в одном и том же сеансе FTP была достигнута производительность 1,313 Мбит/с для Ethernet и только 0,322 Мбит/с для ATM.

В ходе дальнейших исследований авторы обнаружили, что размер буфера, величина MTU (максимальный размер передаваемого блока), максимальный размер сегмента TCP (MSS) и способ передачи данных уровню TCP от слоя сокетов влияли на взаимодействие алгоритма Нейгла и алгоритма отложенного подтверждения (совет 24).

Примечание *MTU (максимальный блок передачи) – это максимальный размер фрейма, который может быть передан по физической сети. Для Ethernet эта величина составляет 1500 байт. Для сети ATM, описанной в работе [Comer and Lin 1995], – 9188 байт.*

Хотя эти результаты были получены для локальной сети ATM и конкретной реализации TCP (SunOS 4.1.1), они применимы и к другим сетям и реализациям.

Самые важные параметры: величина MTU и способ обмена между сокетами и TCP, который в большинстве реализаций, производных от TCP, один и тот же.

Авторы нашли весьма элегантное решение проблемы. Его привлекательность в том, что изменять надо только размер буфера передачи, а размер буфера приема не играет роли. Описанное в работе [Comer and Lin 1995] взаимодействие не имеет места, если размер буфера передачи, по крайней мере, в три раза больше, чем MSS.

Примечание *Смысл этого решения в том, что получателя вынуждают послать информацию об обновлении окна, а, значит, и ACK, предотвращая тем самым откладывание подтверждения и нежелательную интерференцию с алгоритмом Нейгла. Причины обновления информации о размере окна, различны для случаев, когда буфер приема меньше или больше утроенного MSS, но в любом случае обновление посылается.*

Поэтому неинтерактивные приложения всегда должны устанавливать буфер приема не менее чем 3 × MSS. Вспомните совет 7, где сказано, что это следует делать *до* вызова `listen` или `connect`.

Резюме

Производительность TCP в значительной степени зависит от размеров буферов приема и передачи (совет 36). В этом разделе вы узнали, что оптимальный размер буфера для эффективной передачи больших объемов данных равен произведению полосы пропускания на задержку, но на практике это наблюдение не особенно полезно.

Хотя правило произведения применять трудно, есть другое, намного проще. Ему и рекомендуется всегда следовать: размер буфера передачи должен быть, по крайней мере, в три раза больше, чем MSS.

Глава 4. Инструменты и ресурсы

Совет 33. Используйте утилиту ping

Один из самых главных и полезных инструментов отладки сетей и работающих в них приложений – это утилита `ping`. Ее основное назначение – проверить наличие связи между двумя хостами.

Прежде необходимо разъяснить несколько моментов, касающихся `ping`. Во-первых, по словам Майка Муусса, слово «ping» не расшифровывается как «packet internet groper» (проводящий межсетевые пакеты). Своим названием эта программа обязана звуку, который издает сонар, устанавливаемый на подводных лодках. История создания программы `ping` изложена в статье «The Story of the Ping Program» Муусса на Web-странице http://ftp.arl.mil/~mike/ping/html. Там же приведен и ее исходный текст.

Во-вторых, эта утилита не использует ни TCP, ни UDP, поэтому для нее нет никакого хорошо известного порта. Для проверки наличия связи `ping` пользуется функцией эхо-контроля, имеющейся в протоколе ICMP. Помните (совет 14), что, хотя сообщения ICMP передаются в IP-датаграммах, ICMP считается не отдельным протоколом, а частью IP.

Примечание *В RFC 792 [Postel 1981] на первой странице сказано: «ICMP использует базовую поддержку IP, как если бы это был протокол более высокого уровня, однако в действительности ICMP является неотъемлемой частью IP и должен быть реализован в каждом IP-модуле».*

Таким образом, структура пакета, посылаемого `ping`, имеет такой вид, как на рис. 4.1. Показанная на рис. 4.2 ICMP-часть сообщения состоит из восьмибайтного ICMP-заголовка и n байт дополнительной информации.

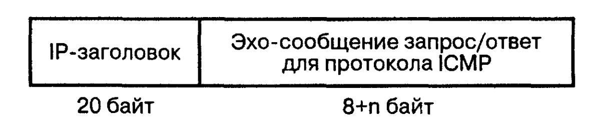


Рис. 4.1. Формат пакета ping

Обычно в качестве значения n – числа дополнительных байтов в пакете `ping` – выбирается 56 (UNIX) или 32 (Windows), но эту величину можно изменить с помощью флагов `-s` (UNIX) или `-l` (Windows).

В некоторых версиях `ping` пользователь может задавать значения дополнительных данных или даже указывать, что они должны генерироваться псевдослучайным образом. По умолчанию в большинстве версий дополнительные данные – это циклически переставляемый по кругу набор байтов.

Примечание *Задание конкретных данных бывает полезно при отладке ошибок, зависящих от данных.*

UNIX-версия ping помещает временной штамп (структуру timeval) в первые восемь байт дополнительных данных (при условии, конечно, что пользователь не задал меньшее количество). Когда программа ping получает эхо-ответ, она использует этот временной штамп для вычисления периода кругового обращения (RTT). Windows-версия ping этого не делает (вывод основан на анализе информации, полученной с помощью программы tcpdump), но в тексте примера ping, поставляемого в составе компилятора Visual C++, этот алгоритм присутствует.

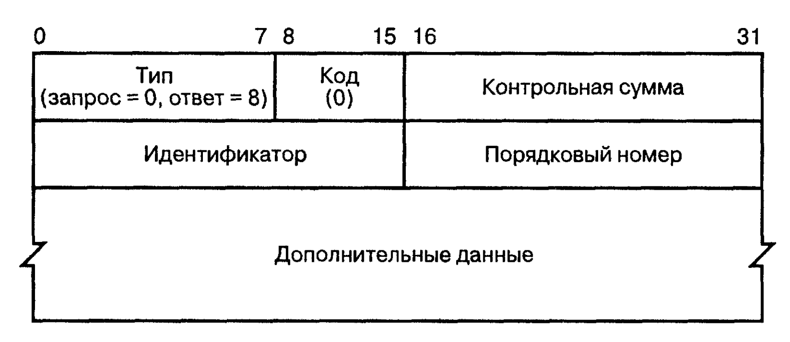


Рис. 4.2. Пакет эхо-сообщения запрос/ответ протокола ICMP

Поскольку поля идентификатор и порядковый номер не задействованы ни в эхо-запросе, ни в эхо-ответе, ping использует их для идентификации полученных ICMP-ответов. Так как для IP-датаграмм нет специального порта, они доставляются каждому процессу, который открыл простой (raw) сокет с указанием протокола ICMP (совет 40). Поэтому ping помещает свой идентификатор процесса в поле идентификатор, так что каждый запущенный экземпляр способен отличить ответы на свои запросы. Таким образом, поле идентификатор в этом случае можно представить как аналог номера порта.

Таким же образом ping помещает в поле порядковый номер значение счетчика для того, чтобы связать ответ с запросом. Именно это значение ping показывает как icmp_seq.

Обычно первое ваше действие при пропадании связи с удаленным хостом – это запуск ping с указанием адреса этого хоста (хост «пингуется»). Предположим, что нужно связаться с хостом A с помощью программы telnet, но соединение не устанавливается из-за истечения тайм-аута. Причин может быть несколько: проблема в сети между двумя хостами, не работает сам хост A, проблема в удаленном стеке TCP/IP или не запущен сервер telnet.

Сначала следует проверить, «пингуя» хост A, что он достижим. Если работа ping проходит нормально, то можно сказать, что с сетью все в порядке, а проблема, вероятно, связана с самим хостом A. Если же невозможно «достучаться» до хоста A с помощью ping, то требуется «пропинговать» ближайший маршрутизатор, чтобы понять, удается ли достичь хотя бы границы собственной сети. Если это получается,

то воспользуйтесь программой traceroute (совет 35), чтобы выяснить, насколько далеко можно продвинуться по маршруту от вашего хоста к хосту А. Часто это помогает идентифицировать сбойный маршрутизатор или хотя бы сделать предположение о месте возникновения ошибки.

Поскольку ping работает на уровне протокола IP, она не зависит от правильности конфигурации TCP или UDP. Поэтому иногда полезно «пропинговать» свой собственный хост, чтобы проверить правильность установки сетевого программного обеспечения. Сначала можно указать ping возвратный адрес localhost (127.0.0.1), чтобы убедиться в работе хотя бы части сетевой поддержки. Если при этом проблем не возникает, то следует «пропинговать» один или несколько сетевых интерфейсов и удостовериться, что они правильно сконфигурированы.

Попробуйте «пропинговать» хост netcom4.netcom.com, который находится от вас в десяти переходах (рис. 4.3).

```
bsd: $ ping netcom4.netcom.com
PING netcom4.netcom.com (199.183.9.104): 56 data bytes
64 bytes from 199.183.9.104: icmp_seq=0 ttl=245 time=598.554 ms
64 bytes from 199.183.9.104: icmp_seq=1 ttl=245 time=550.081 ms
64 bytes from 199.183.9.104: icmp_seq=2 ttl=245 time=590.079 ms
64 bytes from 199.183.9.104: icmp_seq=3 ttl=245 time=530.114 ms
64 bytes from 199.183.9.104: icmp_seq=5 ttl=245 time=480.137 ms
64 bytes from 199.183.9.104: icmp_seq=6 ttl=245 time=540.081 ms
64 bytes from 199.183.9.104: icmp_seq=7 ttl=245 time=580.084 ms
64 bytes from 199.183.9.104: icmp_seq=8 ttl=245 time=490.078 ms
64 bytes from 199.183.9.104: icmp_seq=9 ttl=245 time=560.090 ms
64 bytes from 199.183.9.104: icmp_seq=10 ttl=245 time=490.090 ms
^C                                 завершили ping вручную
--- netcom4.netcom.com ping statistics ---
12 packets transmitted, 10 packets received, 16% packet loss
round-trip min/avg/max/stddev = 480.137/540.939/598.554/40.871 ms
bsd: $
```

Рис. 4.3. Короткий прогон ping

Прежде всего, RTT для разных пакетов мало меняется и остается в пределах 500 мс. Как следует из последней строки, RTT модифицируется в диапазоне от 480,137 мс до 598,554 мс со стандартным отклонением 40,871 мс. Текст слишком рано прерван, чтобы можно было сделать какие-то выводы, но и при более длительном прогоне (около 2 мин) результат существенно не изменится. Так что можно предположить, что нагрузка на сеть постоянная. Значительный разброс RTT – это, как правило, признак изменяющейся загрузки сети. При повышенной загрузке возрастает длина очереди в маршрутизаторе, а вместе с ней – и RTT. При уменьшении загрузки очередь сокращается, что приводит к уменьшению RTT.

Далее из рис. 4.3 видно, что на эхо-запрос ICMP с порядковым номером 4 не пришел ответ. Это означает, что запрос либо ответ был потерян одним из промежуточных маршрутизаторов. По данным сводной статистики, было послано 12 запросов (0–11)

и получено лишь 10 ответов. Один из пропавших ответов имеет порядковый номер 4, второй – 11 (вероятно, он был засчитан как пропавший, поскольку не вовремя прервана работа ping).

Резюме

Утилита ping – это один из важнейших инструментов тестирования связи в сети. Поскольку для ее работы требуется лишь функционирование самых нижних уровней сетевых служб, она полезна для проверки связи в условиях, когда сервисы более высокого уровня, такие как TCP, или программы прикладного уровня типа telnet не работают.

С помощью ping часто удается сделать выводы об условиях в сети, наблюдая за значениями и дисперсией RTT и за числом потерянных ответов.

Совет 34. Используйте программу tcpdump или аналогичное средство

Из всех имеющихся в нашем распоряжении мощных и полезных средств отладки сетевых приложений и поиска неисправностей в сети наиболее интересны сетевые анализаторы (их еще называют сниферами). Традиционно сетевой анализатор – дорогое специализированное устройство, но современные рабочие станции вполне способны выполнять их функции в рамках отдельного процесса.

Сегодня сниферы есть для большинства сетевых операционных систем. Иногда в операционную систему входит снифер, предлагаемый поставщиком (программа snoop в Solaris или программы iptrace/ipreport в AIX), а иногда пользуются программами третьих фирм, например tcpdump.

Из инструментов, предназначенных только для диагностики, сниферы постепенно превратились в средства для исследований и обучения. Например, они постоянно используются для изучения динамики и взаимодействий в сетях. В книгах [Stevens 1994, Stevens 1996] рассказано, как использовать tcpdump, чтобы разобраться в работе сетевых протоколов. Наблюдая за данными, которые посылает протокол, вы можете глубже понять его функционирование на практике, а заодно увидеть, когда некоторая конкретная реализация работает не в соответствии со спецификацией.

В этом разделе будет рассмотрена утилита tcpdump. Как уже отмечалось, есть и другие программно реализованные сетевые анализаторы. Некоторые из них лучше форматируют выходные данные, но у tcpdump есть одно неоспоримое преимущество – она работает практически во всех UNIX-системах и в Windows. Поскольку исходные тексты tcpdump опубликованы, ее можно при необходимости адаптировать для специальных целей или перенести на новую платформу.

Код tcpdump вы можете найти на сайте http://www-nrg.ee.lbl.gov/nrg.html, а исходные тексты и исполняемый код для Windows WinDump – http://netgroup-serv.polito.it/windump.

Как работает tcpdump

Посмотрим, как работает программа tcpdump и на каком уровне протоколов она перехватывает пакеты. Как и большинство сетевых анализаторов, tcpdump состоит

из двух компонент: первая работает в ядре и занимается перехватом и, возможно, фильтрацией пакетов, а вторая действует в адресном пространстве пользователя и определяет интерфейс пользователя, а также выполняет форматирование и фильтрацию пакетов, если последнее не делается ядром.

Пользовательская компонента `tcpdump` взаимодействует с компонентой в ядре при помощи библиотеки `libpcap` (библиотека для перехвата пакетов), которая абстрагирует системно-зависимые детали общения с канальным уровнем стека протоколов. Например, в системах на основе BSD `libpcap` взаимодействует с *пакетным фильтром* BSD (BSD packet filter – BPF) [McCanne and Jacobson 1993]. BPF исследует каждый пакет, проходящий через канальный уровень, и сопоставляет его с фильтром, заданным пользователем. Если пакет удовлетворяет критерию фильтрации, то его копия помещается в выделенный ядром буфер, который ассоциируется с данным фильтром. Когда буфер заполняется или истекает заданный пользователем тайм-аут, содержимое буфера передается приложению с помощью `libpcap`.

Этот процесс изображен на рис. 4.4. Показано, как `tcpdump` и любая другая программа считывают необработанные пакеты с помощью BPF, а также изображено еще одно приложение, читающее данные из стека TCP/IP, как обычно.

Примечание *Хотя на этом рисунке и `tcpdump`, и программа используют библиотеку `libpcap`, можно напрямую общаться с BPF или иным интерфейсом, о чем будет сказано ниже. Достоинство `libpcap` в том, что она предоставляет системно-независимые средства доступа к необработанным пакетам. В настоящее время эта библиотека поддерживает BPF; интерфейс канального провайдера (data link provider interface – DLPI; систему SunOS NIT; потоковую NIT; сокеты типа SOCK_PACKET, применяемые в системе Linux; интерфейс `snoop` (IRIX) и разработанный в Стэнфордском университете интерфейс `enet`. В дистрибутив `WinDump` входит также версия `libpcap` для Windows.*

Обратите внимание, что BPF перехватывает сетевые пакеты на уровне драйвера устройства, то есть сразу после того, как они считаны с носителя. Это не то же самое, что чтение из простого сокета. В ситуации с простым сокетом вы получаете IP-датаграммы, уже обработанные уровнем IP и переданные непосредственно приложению, минуя транспортный уровень (TCP или UDP). Об этом рассказывается в совете 40.

Начиная с версии 2.0, архитектура `WinDump` очень напоминает используемую в системах BSD. Эта программа пользуется специальным NDIS-драйвером (NDIS – Network Driver Interface Specification – спецификация стандартного интерфейса сетевых адаптеров), предоставляющим совместимый с BPF фильтр и интерфейс. В архитектуре `WinDump` NDIS-драйвер фактически представляет собой часть стека протоколов, но функционирует он так же, как показано на рис. 4.4, только надо заменить BPF на пакетный драйвер NDIS.

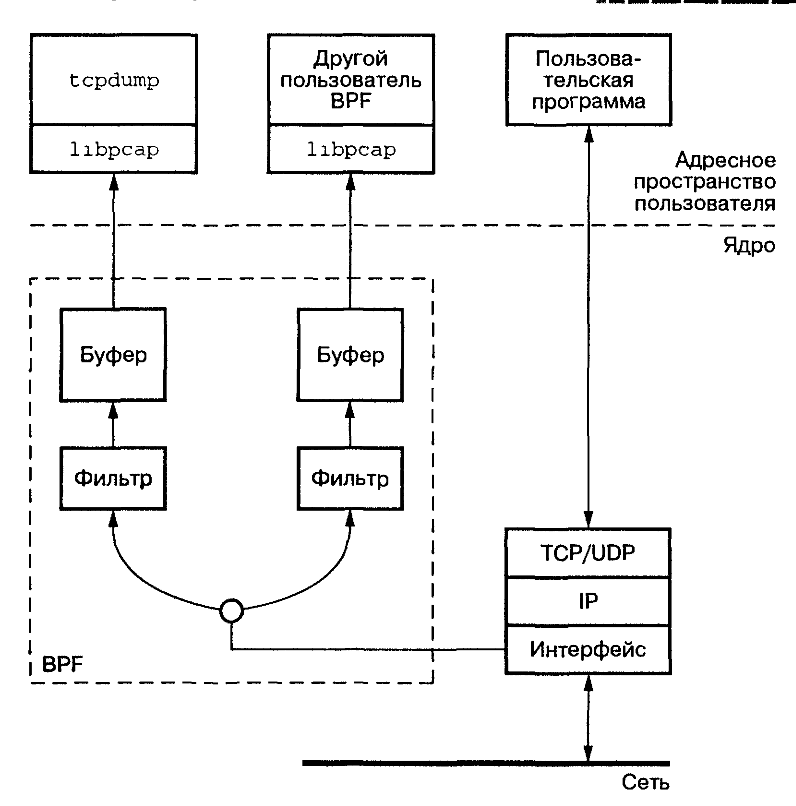


Рис. 4.4. Перехват пакетов с помощью BPF

Другие операционные системы используют несколько иные механизмы. В системах, производных от SVR4, для доступа к простым сокетам применяется интерфейс DLPI [Unix International 1991]. DLPI – это не зависящий от протокола, основанный на системе STREAMS [Ritchie 1984] интерфейс к канальному уровню. С помощью DLPI можно напрямую получить доступ к канальному уровню, но по соображениям эффективности обычно вставляют в поток STREAMS-модули `pfmod` и `bufmod`. Модуль `bufmod` предоставляет услуги по буферизации сообщений и увеличивает эффективность за счет ограничения числа контекстных переключений, требуемых для доставки данных.

Примечание *Это аналогично чтению полного буфера из сокета вместо побайтного чтения.*

Модуль `pfmod` – это фильтр, аналогичный BPF. Поскольку он несовместим с фильтром BPF, `tcpdump` вставляет этот модуль в поток, а фильтрацию выполняет в пространстве пользователя. Это не столь эффективно, как при использовании BPF, так как в пространство пользователя приходится передавать каждый пакет, даже если он не нужен программе `tcpdump`.

На рис. 4.5 показаны tcpdump без модуля pfmod и приложение, которое получает необработанные пакеты с использованием находящегося в ядре фильтра.

На рис. 4.5 также представлены приложения, пользующиеся библиотекой libpcap, но, как и в случае BPF, это необязательно. Для отправки сообщений непосредственно в поток и получения их обратно можно было бы воспользоваться вызовами getmsg и putmsg. Книга [Rago 1993] – отличный источник информации о программировании системы STREAMS, DLPI и системных вызовах getmsg и putmsg. Более краткое обсуждение вопроса можно найти в главе 33 книги [Stevens 1998].

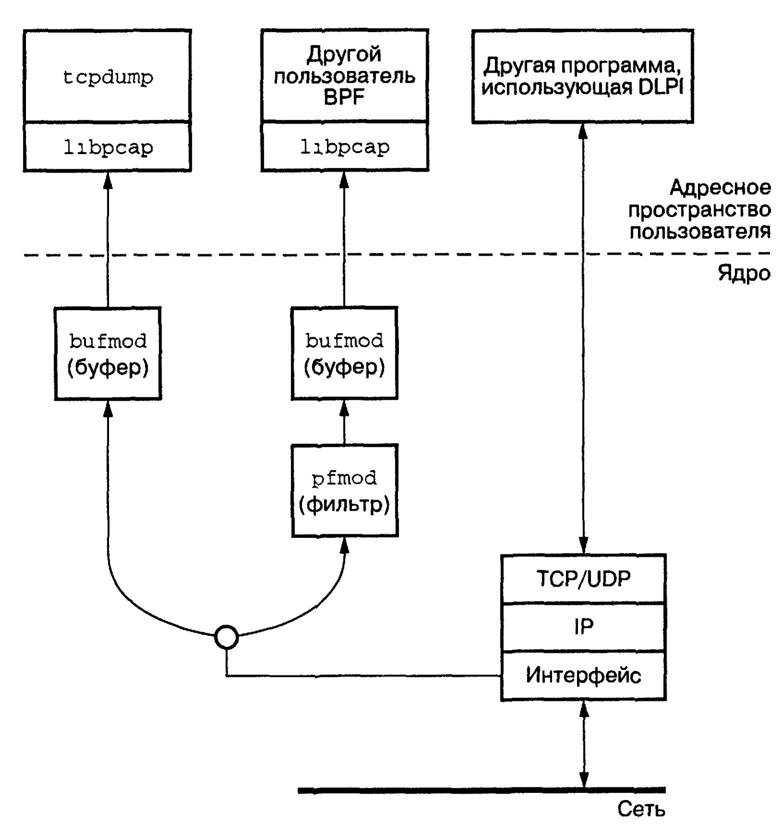


Рис. 4.5. Перехват пакетов с помощью DLPI

Наконец, есть еще и архитектура Linux. В этой системе доступ к необработанным сетевым пакетам производится через интерфейс сокетов типа SOCK_PACKET. Для использования этого простого механизма надо открыть подобный сокет, привязать к нему требуемый сетевой интерфейс, включить режим пропускания всех пакетов (promiscuous mode) и читать из сокета.

Примечание *Начиная с версии 2.2 ядра Linux, рекомендуется несколько другой интерфейс, но последняя версия libpcap по-прежнему поддерживает описанный выше.*

Например, строка

```
s = socket( AF_INET, SOCK_PACKET, htons( ETH_P_ALL ) );
```

открывает сокет, предоставляющий доступ ко всем Ethernet-пакетам. В качестве третьего параметра можно также указать ETH_P_IP (пакеты IP), ETH_P_IPV6 (пакеты IPv6) или ETH_P_ARP (пакеты ARP). Будем считать, что этот интерфейс аналогичен простым сокетам (SOCK_RAW), только доступ производится к канальному, а не сетевому (IP) уровню.

К сожалению, несмотря на простоту и удобство этого интерфейса, он не очень эффективен. В отличие от обычных сокетов, ядро в этом случае не осуществляет никакой буферизации, так что каждый пакет доставляется приложением сразу после поступления. Отсутствует также фильтрация на уровне ядра (если не считать параметра ETH_P_*). Поэтому фильтровать приходится на прикладном уровне, а это означает, что приложение должно получать все пакеты без исключения.

Использование tcpdump

Прежде всего для использования tcpdump надо получить разрешение. Поскольку применение сетевых анализаторов небезопасно, по умолчанию tcpdump конфигурируется с полномочиями суперпользователя root.

Примечание *К системе Windows это не относится. Коль скоро NDIS-драйвер для перехвата пакетов установлен, воспользоваться программой WinDump может любой.*

Во многих случаях лучше дать возможность всем пользователям работать с программой tcpdump, не передавая им полномочия суперпользователя. Это делается по-разному, в зависимости от версии UNIX и документировано в руководстве по tcpdump. В большинстве случаев надо либо предоставить всем права на чтение из сетевого интерфейса, либо сделать tcpdump setuid-программой.

Проще всего вызвать tcpdump вообще без параметров. Тогда она будет перехватывать все сетевые пакеты и выводить о них информацию. Однако полезнее указать какой-нибудь фильтр, чтобы видеть только нужные пакеты и не отвлекаться на остальные. Например, если требуются лишь пакеты, полученные от хоста bsd или отправленные ему, то можно вызвать tcpdump так:

```
tcpdump host bsd
```

Если же нужны пакеты, которыми обмениваются хосты bsd и sparc, то можно использовать такой фильтр:

```
host bsd and host sparc
```

или сокращенно –

```
host bsd and sparc
```

Язык для задания фильтров достаточно богат и позволяет фильтровать, например, по следующим атрибутам:

- протокол;
- хост отправления и/или назначения;
- сеть отправления и/или назначения;
- Ethernet-адрес отправления и/или назначения;
- порт отправления и/или назначения;
- размер пакета;
- пакеты, вещаемые на всю локальную сеть или на группу (как в Ethernet, так и в IP);
- пакет, используемый в качестве шлюза указанным хостом.

Кроме того, можно проверять конкретные биты или байты в заголовках протоколов. Например, чтобы отбирать только TCP-сегменты, в которых выставлен бит срочных данных, следует использовать фильтр

```
tcp[ 13 ] & 16
```

Чтобы понять последний пример, надо знать, что четвертый бит четырнадцатого байта заголовка TCP – это бит срочности.

Поскольку разрешается использовать булевские операторы `and` (или `&&`), `or` (или `||`) и `not` (или `!`) для комбинирования простых предикатов, можно задавать фильтры произвольной сложности. Ниже приведен пример фильтра, отбирающего ICMP-пакеты, приходящие из внешней сети:

```
icmp and not src net localnet
```

Примеры более сложных фильтров рассматриваются в документации по `tcpdump`.

Выходная информация, формируемая tcpdump

Информация, выдаваемая программой `tcpdump`, зависит от протокола. Рассмотрим несколько примеров, которые помогут составить представление о том, что можно получить от `tcpdump` для наиболее распространенных протоколов. Документация, поставляемая вместе с программой, содержит исчерпывающие сведения о формате выдачи.

Первый пример – это трассировка сеанса по протоколу SMTP (Simple Mail Transfer Protocol – простой протокол электронной почты), то есть процедура отправки электронного письма. Распечатка на рис. 4.6 в точности соответствует выдаче `tcpdump` – только добавлены номера строк, напечатанные курсивом, удалено имя домена хоста `bsd` и перенесены длинные строки, не уместившиеся на странице.

Для получения трассировки послано письмо пользователю с адресом в домене `gte.net`. Таким образом, адрес имел вид `user@gte.net`.

Строки 1–4 относятся к поиску адреса SMTP-сервера, обслуживающего домен `gte.net`. Это пример выдачи, генерируемой `tcpdump` для запросов и ответов сервиса DNS. В строке 1 `bsd` запрашивает у сервера имен своего сервис-провайдер (`ns1.ix.netcom.com`) имя или имена почтового сервера `gte.net`. В первом поле находится временной штамп пакета (12:54:32.920881). Поскольку разрешающая способность таймера на машине `bsd` составляет 1 мкс, показано шесть десятичных знаков. Вы видите, что пакет ушел из порта 1067 на `bsd` в порт 53 (`domain`) на машине `ns1`. Далее, дается информация о данных в пакете. Первое поле (45801) –

```
1  12:54:32.920881 bsd.1067 > ns1.ix.netcom.com.domain:
       45801+ MX? gte.net. (25)
2  12:54:33.254981 ns1.ix.netcom.com.domain > bsd.1067:
       45801 5/4/9 (371) (DF)
3  12:54:33.256127 bsd.1068 > ns1.ix.netcom.com.domain:
       45802+ A? mtapop2.gte.net. (33)
4  12:54:33.534962 ns1.ix.netcom.com.domain > bsd.1068:
       45802 1/4/4 (202) (DF)
5  12:54:33.535737 bsd.1059 > mtapop2.gte.net.smtp:
       S 585494507:585494507(0) win 16384
       <mss 1460,nop,wscale 0,nop,nop,
       timestamp 6112 0> (DF)
6  12:54:33.784963 mtapop2.gte.net.smtp > bsd.1059:
       S 1257159392:1257159392(0) ack 585494509 win 49152
       <mss 1460,nop,wscale 0,nop,nop,
       timestamp 7853753 6112> (DF)
7  12:54:33.785012 bsd.1059 > mtapop2.gte.net.smtp:
       . ack 1 win 17376 <nop,nop,
       timestamp 6112 7853753> (DF)
8  12:54:34.235066 mtapop2.gte.net.smtp > bsd.1059:
       P 1:109(108) ack 1 win 49152
       <nop,nop,timestamp 7853754 6112> (DF)
9  12:54:34.235277 bsd.1059 > mtapop2.gte.net.smtp:
       P 1:19(18) ack 109 win 17376
       <nop,nop,timestamp 6113 7853754> (DF)
       14 строк опущено
24 12:54:36.675105 bsd.1059 > mtapop2.gte.net.smtp:
       F 663:663(0) ack 486 win 17376
       <nop,nop,timestamp 6118 7853758> (DF)
25 12:54:36.685080 mtapop2.gte.net.smtp > bsd.1059:
       F 486:486(0) ack 663 win 49152
       <nop,nop,timestamp 7853758 6117> (DF)
26 12:54:36.685126 bsd.1059 > mtapop2.gte.net.smtp:
       . ack 487 win 17376
       <nop,nop,timestamp 6118 7853758> (DF)
27 12:54:36.934985 mtapop2.gte.net.smtp > bsd.1059:
       F 486:486(0) ack 664 win 49152
       <nop,nop,timestamp 7853759 6118> (DF)
28 12:54:36.935020 bsd.1059 > mtapop2.gte.net.smtp:
       . ack 487 win 17376
       <nop,nop,timestamp 6118 7853759> (DF)
```

Рис. 4.6. Трассировка SMTP-сеанса с включением обмена по протоколам DNS и TCP

это номер запроса, используемый функциями разрешения имен на bsd для сопоставления ответов с запросами. Знак «+» означает, что функция разрешения задает опрос DNS-сервером других серверов, если у него нет информации об ответе. Строка «MX?» показывает, что это запрос о записи почтового обмена для сети, имя которой стоит в следующем поле (gte.net). Строка «(25)» свидетельствует о том, что длина запроса – 25 байт.

Строка 2 – это ответ на запрос в строке 1. Число 45801 – это номер запроса, к которому относится ответ. Следующие три поля, разделенные косой чертой, – количество записей в ответе, записей от сервера имен (полномочного агента) и прочих записей. Строка «(371)» показывает, что ответ содержит 371 байт. И, наконец, строка «(DF)» означает, что в IP-заголовке ответа был поднят бит «Don't fragment» (не фрагментировать). Итак, эти две строки иллюстрируют использование системы DNS для поиска обработчиков почты (об этом кратко упоминалось в совете 29).

Если в двух первых строках было выяснено имя обработчика почты для сети `gte.net`, то в двух последующих выясняется егоПрограмма tcpdump IP-адрес. «A?» в строке 3 указывает, что это запрос IP-адреса хоста `mtapop2.gte.net` – одного из почтовых серверов компании GTE.

Строки 5–28 содержат детали обмена по протоколу SMTP. Процедура трехстороннего квитирования между хостами `bsd` и `mtapop2` начинается в строке 5 и заканчивается строкой 7. Первое поле после временного штампа и имен хостов – это поле *`flags`*. «S» в строке 5 указывает, что в сегменте установлен флаг SYN. Другие возможные значения флага: «F» (FIN), «U» (URG), «P» (PUSH), «R» (RST) и «.» (нет флагов). Далее идут порядковые номера первого и последнего байтов, а за ними в скобках – число байтов данных. Эти поля могут вызвать некоторое недоумение, так как «порядковый номер последнего» – это первый неиспользованный порядковый номер, но только в том случае, когда в пакете есть данные. Удобнее всего считать, что первое число – это порядковый номер первого байта в сегменте (SYN или информационном), а второе – порядковый номер первого байта плюс число байтов данных в сегменте. Следует отметить, что по умолчанию показываются реальные порядковые номера для SYN-сегментов и смещения – для последующих сегментов (так удобнее следить). Это поведение можно изменить с помощью опции `-S` в командной строке.

Во всех сегментах, кроме первого SYN, имеется поле *ACK*, показывающее, какой следующий порядковый номер ожидает отправитель. Это поле (в виде `ack` *`nnn`*), как и раньше, по умолчанию содержит смещение относительно порядкового номера, указанного в сегменте SYN.

За полем ACK идет поле *`window`*. Это количество байтов данных, которое готов принять удаленный хост. Обычно оно отражает объем свободной памяти в буферах соединения.

И, наконец, в угловых скобках указаны опции TCP. Основные опции рассматриваются в RFC 793 [Postel 1981b] и RFC 1323 [Jacobson et al. 1992]. Они обсуждаются также в книге [Stevens 1994], а их полный перечень можно найти на Web-странице http://www.isi.edu/in-notes/iana/assignments/tcp-parameters.

В строках 8–23 показан диалог между программой `sendmail` на `bsd` и SMTP-сервером на машине `mtapop2`. Большая часть этих строк опущена. Строки 24–28 отражают процедуру разрыва соединения. Сначала `bsd` посылает FIN в строке 24, затем приходит FIN от `mtapop2` (строка 25). Заметьте, что в строке 27 `mtapop2` повторно посылает FIN. Это говорит о том, что хост не получил от `bsd` подтверждения ACK на свой первый FIN, и еще раз подчеркивает важность состояния TIME-WAIT (совет 22).

Теперь посмотрим, что происходит при обмене UDP-датаграммами. С помощью клиента udphelloc (совет 4) следует послать один нулевой байт в порт сервера времени дня в домене netcom.com:

```
bsd: $ udphelloc netcom4.netcom.com daytime
Thu Sep 16 15:11:49 1999
bsd: $
```

Хост netcom4 возвращает дату и время в UDP-датаграмме. Программа tcpdump печатает следующее:

```
18:12:23.130009 bsd.1127 > nectom4.netcom.com.daytime: udp 1
18:12:23.389284 nectom4.netcom.com.daytime > bsd.1127: udp 26
```

Отсюда видно, что bsd послал netcom4 UDP-датаграмму длиной один байт, а netcom4 ответил датаграммой длиной 26 байт.

Протокол обмена ICMP-пакетами аналогичен. Ниже приведена трассировка одного запроса, генерируемого программой ping с хоста bsd на хост netcom4:

```
1  06:21:28.690390 bsd > netcom4.netcom.com: icmp: echo request
2  06:21:29.400433 netcom4.netcom.com > bsd: icmp: echo reply
```

Строка icmp: означает, что это ICMP-датаграмма, а следующий за ней текст описывает тип этой датаграммы.

Один из недостатков tcpdump – это неполная поддержка вывода собственно данных. Часто во время отладки сетевых приложений необходимо знать, какие данные посылаются. Эту информацию можно получить, задав в командной строке опции -s и -x, но данные будут выведены только в шестнадцатеричном формате. Опция -x показывает, что содержимое пакета нужно выводить в шестнадцатеричном виде. Опция -s сообщает, сколько данных из пакета выводить. По умолчанию tcpdump выводит только первые 68 байт (в системе SunOS NIT – 96 байт). Этого достаточно для заголовков большинства протоколов. Повторим предыдущий пример, касающийся UDP, но здесь нужно выводить также следующие данные:

```
tcpdump -x -s 100 -l
```

После удаления строк, относящихся к DNS, и исключения имени домена из адреса хоста bsd получается следующий результат:

```
1  12:57:53.299924 bsd.1053 > netcom4.netcom.com.daytime: udp 1
                   4500 001d 03d4 0000 4011 17a1 c7b7 c684
                   c7b7 0968 041d 000d 0009 9c56 00
2  12:57:53.558921 netcom4.netcom.com.daytime > bsd.1053: udp 26
                   4500 0036 f0c8 0000 3611 3493 c7b7 0968
                   c7b7 c684 000d 041d 0022 765a 5375 6e20
                   5365 7020 3139 2030 393a 3537 3a34 3220
                   3139 3939 0a0d
```

Последний байт в первом пакете – это нулевой байт, который udphelloc посылает хосту netcom4. Последние 26 байт второго пакета – это полученный ответ. Интерпретировать приведенные в нем шестнадцатеричные цифры довольно трудно.

Авторы tcpdump не хотели давать ASCII-представление данных, так как полагали, что это упростит кражу паролей для технически неподготовленных лиц. Теперь многие считают, что широкое распространение программ для кражи паролей сделало это опасение неактуальным. Поэтому есть основания полагать, что в последующие версии tcpdump будет включена поддержка вывода в коде ASCII*.

А пока многие сетевые программисты упражняются в написании фильтров, преобразующих выдачу tcpdump в код ASCII. Несколько подобных программ есть в Internet. Показанный в листинге 4.1 сценарий Perl запускает tcpdump, перенаправляет ее вывод к себе и перекодирует данные в ASCII.

Листинг 4.1. Perl-сценарий для фильтрации выдачи tcpdump

tcpd

```
#! /usr/bin/perl5
$tcpdump = "/usr/sbin/tcpdump";
open( TCPD, "$tcpdump @ARGV |" ) ||
die "не могу запустить tcpdump: \$!\\n";
$| = 1;
while ( <TCPD> )
{
   if ( /^\t/ )
   {
      chop;
      $str = $_;
      $str =~ tr / \t//d;
      $str = pack "H*" , $str;
      $str =~ tr/\x0-\x1f\x7f-\xff/./;
      printf "\t%-40s\t%s\n", substr( $_, 4 ), $str;
   }
   else
   {
      print;
   }
}
```

tcpd

Если еще раз прогнать последний пример, но вместо tcpdump использовать tcpd, то получится следующее:

```
1  12:58:56.428052 bsd.1056 > netcom4.netcom.com.daytime: udp 1
       4500 001d 03d7 0000 4011 179e c7b7 c684 E.......@.......
       c7b7 0968 041d 000d 0009 9c56 00 ...h. .....S.
2  12:58:56.717128 netcom4.netcom.com.daytime > bsd.1053: udp 26
       4500 0036 10f1 0000 3611 146b c7b7 0968  E..6....6..k...h
       c7b7 c684 000d 0420 0022 7656 5375 6e20  ....... ."rVSun
       5365 7020 3139 2030 393a 3538 3a34 3620  Sep 19 09:58:46
       3139 3939 0a0d                           1999..
```

* Начиная с версии 3.5 tcpdump позволяет выводить и ASCII-представление. Для этого надо одновременно указать опции -X и -x – *Прим. автора.*

Резюме

Программа `tcpdump` – это незаменимый инструмент для изучения того, что происходит в сети. Если знать, что в действительности посылается или принимается «по проводам», то трудные, на первый взгляд, ошибки удается легко найти и исправить. Эта программа представляет собой также важный инструмент для исследований динамики сети, а равно средство обучения. В последнем качестве она широко применяется в книгах серии «TCP/IP Illustrated», написанных Стивенсом.

Совет 35. Применяйте программу traceroute

Утилита `traceroute` – это важный инструмент для нахождения ошибок маршрутизации, изучения трафика в Internet и исследования топологии сети. Как и многие другие распространенные сетевые инструменты, `traceroute` была разработана коллективом лаборатории Лоренса Беркли в Университете Калифорнии.

Примечание *В комментариях к исходному тексту Ван Джекобсон, автор программы `traceroute`, пишет: «Я пытался найти ошибку в работе алгоритма маршрутизации в течение 48 бессонных часов, и этот код родился как-то сам собой».*

Идея `traceroute` проста. Программа пытается определить маршрут между двумя хостами в сети, заставляя каждый промежуточный маршрутизатор посылать ICMP-сообщение об ошибке хосту-отправителю. Далее об этом механизме будет сказано подробнее. Сначала нужно несколько раз запустить программу и посмотреть, что она выдает. Проследим маршрут между хостом `bsd` и компьютером в Университете города Тампа на юге Флориды (рис. 4.7). Как обычно, перенесены строки, не умещающиеся на странице.

Число слева в каждой строке – это номер промежуточного узла. За ним идет имя хоста или маршрутизатора в этом узле и далее – IP-адрес узла. Если узнать имя не удается, то `traceroute` печатает только IP-адрес. Такая ситуация наблюдается в узле 13. Как видно, по умолчанию программа пыталась определить имя хоста или маршрутизатора трижды, а три числа, следующие за IP-адресом, – это периоды кругового обращения (RTT) для каждой из трех попыток. Если при очередной попытке на запрос никто не отвечает или ответ теряется, то вместо времени печатается «*».

Хотя компьютер `ziggy.usf.edu` расположен в соседнем городе, в Internet между ними находится 14 узлов. Сначала данные проходят через два маршрутизатора в Тампе, относящихся к сети `netcom.net` (это сервис-провайдер, через которого `bsd` выходит в Internet), потом еще через два маршрутизатора, а затем через маршрутизатор `netcom.net` в узле MAE-EAST (узел 5) в сеть, находящуюся в Вашингтоне, округ Колумбия. Узел MAE-EAST – это точка пересечения сетей, в которой сервис-провайдеры передают друг другу Internet-трафик. Далее покидает узел MAE-EAST и попадает в сеть `sprintlink.net`. От маршрутизатора сети Sprintlink в узле MAE-EAST он пролегает вдоль восточного побережья до домена `usf.edu` (узел 13). И наконец на шаге 14 маршрут подходит к компьютеру ziggy.

```
bsd: $ traceroute ziggy. usf. edu
traceroute to ziggy. usf. edu (131. 247. 1. 40), 30 hops max,
                     40 byte packets
  1 tam-fl-pm8. netcom. net (163. 179. 44. 15)
                     128. 960 ms  139. 230 ms  129. 483 ms
  2 tam-fl-gw1. netcom. net (163. 179. 44. 254)
                     139. 436 ms  129.226 ms  129.570 ms
  3 hl-0.mig-fl-gw1.netcom.net (165.236.144.110)
                     279.582 ms  199.325 ms  289.611 ms
  4 a5-0-0-7.was-dc-gw1.netcom.net (163.179.235.121)
                     179.505 ms  229.543 ms  179.422 ms
  5 hl-0.mae-east.netcom.net (163.179.220.182)
                     189.258 ms  179.211 ms  169.605 ms
  6 sl-mae-e-f0-0.sprintlink.net (192.41.177.241)
                     189.999 ms  179.399 ms  189.472 ms
  7 sl-bb4-dc-1-0-0.sprintlink.net (144.228.10.41)
                     180.048 ms  179.388 ms  179.562 ms
  8 sl-bb10-rly-2-3.sprintlink.net (144.232.7.153)
                     199.433 ms 179.390 ms 179.468 ms
  9 sl-bb11-rly-9-0.sprintlink.net (144.232.0.46)
                     199.259 ms 189.315 ms 179.459 ms
 10 sl-bb10-orl-1-0.sprintlink.net (144.232.9.62)
                     189.987 ms 199.508 ms 219.252 ms
 11 sl-gw3-orl-4-0-0.sprintlink.net (144.232.2.154)
                     219.307 ms 209.382 ms 209.502 ms
 12 sl-usf-1-0-0.sprintlink.net (144.232.154.14)
                     209.518 ms 199.288 ms 219.495 ms
 13 131.247.254.36 (131.247.254.36) 209.318ms 199.281ms 219.588ms
 14 ziggy.usf.edu (131.247.1.40) 209.591 ms * 210.159 ms
```

Рис. 4.7. Маршрут до хоста ziggy.usf.edu, прослеженный traceroute

Посмотрим, как далеко от bsd отстоит Калифорнийский университет в Лос-Анджелесе. Понятно, что географически он находится на другом конце страны, в Калифорнии. А если выполнить traceroute до хоста panther в Калифорнийском университете, то получится результат, показанный на рис. 4.8.

На этот раз маршрут проходит только через 13 промежуточных узлов и достигает домена ucla.edu на шаге 11. Таким образом, топологически bsd ближе к Калифорнийскому университету, чем к Университету на юге Флориды.

Примечание *Университет Чепмена, расположенный также вблизи Лос-Анджелеса, находится всего в девяти промежуточных шагах от bsd. Это связано с тем, что домен chapman.edu, как и bsd, подключен к Internet через сеть netcom.net, и весь трафик проходит по этой опорной сети.*

Как работает traceroute

А теперь разберемся, как работает traceroute. Вспомним (совет 22), что в IP-датаграмме есть поле TTL, которое уменьшается на единицу каждым промежуточным

```
bsd: $ traceroute panther.cs.ucla.edu
traceroute to panther.cs-ucla.edu (131.179.128.25),
                     30 hops max, 40 byte packets
  1 tam-fl-pm8.netcom.net (163.179.44.15)
                     148.957 ms 129.049 ms 129.585 ms
  2 tam-fl-gw1.netcom.net (163.179.44.254)
                     139.435 ms 139.258 ms 139.434 ms
  3 h1-0.mig-fl-gw1.netcom.net (165.236.144.110)
                     139.538 ms 149.202 ms 139.488 ms
  4 a5-0-0-7.was-dc-gw1.netcom.net (163.179.235.121)
                     189.535 ms 179.496 ms 168.699 ms
  5 h2-0.mae-east.netcom.net (163.179.136.10)
                     180.040 ms 189.308 ms 169.479 ms
  6 cpe3-fddi-0.Washington.cw.net (192.41.177.180)
                     179.186 ms 179.368 ms 179.631 ms
  7 core5-hssi6-0-0.Washington.cw.net (204.70.1.21)
                     199.268 ms 179.537 ms 189.694 ms
  8 corerouter2.Bloomington.cw.net (204.70.9.148)
                     239.441 ms 239.560 ms 239.417 ms
  9 bordercore3.Bloomington.cw.net (166.48.180.1)
                     239.322 ms 239.348 ms 249.302 ms
 10 ucla-internet-t-3.Bloomington.cw.net (166.48.181.254)
                     249.989 ms 249.384 ms 249.662 ms
 11 cbn5-t3-1.cbn.ucla.edu (169.232.1.34)
                     258.756 ms 259.370 ms 249.487 ms
 12 131.179.9.6 (131.179.9.6) 249.457 ms 259.238 ms 249.666 ms
 13 Panther.CS.UCLA.EDU (131.179.128.25) 259.256 ms 259.184 ms *
bsd: $
```

Рис. 4.8. Маршрут до хоста panther.cs.ucla.edu, прослеженный traceroute

маршрутизатором. Когда маршрутизатор получает датаграмму, у которой в поле TTL находится единица (или нуль), он отбрасывает ее и посылает отправителю ICMP-сообщение «истекло время в пути».

Программа `traceroute` использует это свойство. Сначала она посылает получателю UDP-датаграмму, в которой TTL установлено в единицу. Когда датаграмма доходит до первого маршрутизатора, тот определяет, что поле TTL равно единице, отбрасывает датаграмму и посылает отправителю ICMP-сообщение. Так вы узнаете адрес первого промежуточного узла (из поля «адрес отправителя» в заголовке ICMP). И `traceroute` пытается выяснить его имя с помощью функции `gethostbyaddr`. Чтобы получить информацию о втором узле, `traceroute` повторяет процедуру, на этот раз установив TTL равным двум. Маршрутизатор в первом промежуточном узле уменьшит TTL на единицу и отправит датаграмму дальше. Но второй маршрутизатор определит единицу в поле TTL, отбросит датаграмму и пошлет ICMP-сообщение отправителю. Повторяя эти действия, но увеличивая каждый раз значение TTL, `traceroute` может построить весь маршрут от отправителя к получателю.

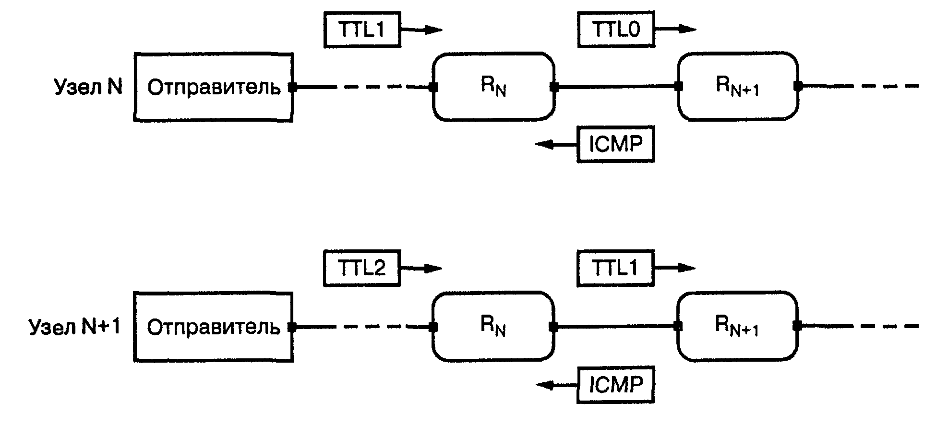


Рис. 4.9. Маршрутизатор N ошибочно переправляет датаграмму с TTL, равным нулю

Когда датаграмма с достаточно большим начальным значением TTL наконец доходит до получателя, TTL будет равно единице, но, поскольку дальше переправлять датаграмму некуда, стек TCP/IP попытается доставить ее ожидающему приложению. Однако `traceroute` установлено в качестве порта назначения такое значение, которое вряд ли кем-то используется, поэтому хост-получатель вернет ICMP-сообщение «порт недоступен». Получив такое сообщение, `tracerout` определяет, что конечный получатель обнаружен, и трассировку можно завершить.

Поскольку протокол UDP ненадежен (совет 1), не исключена возможность потери датаграмм. Поэтому `traceroute` пытается «достучаться» до каждого промежуточного хоста или маршрутизатора несколько раз, то есть посылает несколько датаграмм с одним и тем же значением TTL. По умолчанию делается три попытки, но это можно изменить с помощью опции `-q`.

Кроме того, `tracerout` нужно определить, сколько времени ждать ICMP-сообщения после каждой попытки. По умолчанию время ожидания – 5 с, но это значение можно изменить с помощью опции `-w`. Если в течение этого времени ICMP-сообщение не получено, то вместо значения RTT печатается звездочка (*).

В описанном процессе могут быть некоторые трудности: `traceroute` полагается на то, что маршрутизаторы будут, как положено, отбрасывать IP-датаграммы, в которых TTL равно единице, и посылать при этом ICMP-сообщение «истекло время в пути». К сожалению, некоторые маршрутизаторы таких сообщений не посылают, и тогда печатаются звездочки. Есть также маршрутизаторы, которые посылают сообщение, но с тем значением TTL, которое обнаружили во входящей датаграмме. Поскольку оно оказалось равным нулю, то датаграмма будет отброшена первым же узлом на обратном пути (если, конечно, это не случилось на первом шаге). Результат точно такой же, как если бы ICMP-сообщение не посылалось вовсе.

Некоторые маршрутизаторы ошибочно переправляют далее датаграммы, в которых TTL равно нулю. Если такое происходит, то следующий маршрутизатор, например N + 1, отбросит датаграмму и вернет ICMP-сообщение «истекло время в пути». На дальнейшей итерации маршрутизатор N + 1 получит датаграмму со значением TTL, равным единице, и вернет обычное ICMP-сообщение. Таким

образом, маршрутизатор N + 1 появится дважды: первый раз в результате ошибки предыдущего маршрутизатора, а второй – после корректного отбрасывания датаграммы с истекшим временем работы. Такая ситуация изображена на рис. 4.9, а ее видимое проявление – в строках, соответствующих узлам 5 и 6 на рис. 4.10.

```
bsd: $ traceroute syrup.hill.com
traceroute to syrup.hill.corn (208.162.106.3),
                    30 hops max, 40 byte packets
 1  tam-fl-pm5.netcom.net (163.179.44.11)
                    129.120 ms  139.263 ms  129.603 ms
 2  tam-fl-gw1.netcom.net (163.179.44.254)
                    129.584 ms  129.328 ms  149.578 ms
 3  h1-0.mig-fl-gw1.netcom.net (165.236.144.110)
                    219.595 ms  229.306 ms  209.602 ms
 4  a5-0-0-7.was-dc-gw1.netcom.net (163.179.235.121)
                    179.248 ms  179.521 ms  179.694 ms
 5  h2-0.mae-east.netcom.net (163.179.136.10)
                    179.274 ms  179.325 ms  179.623 ms
 6  h2-0.mae-east.netcom.net (163.179.136.10)
                    169.443 ms  199.318 ms  179.601 ms
 7  cpe3-fddi-0.washington.cw.net (192.41.177.180) 189.529 ms
                    core6-serial5-1-0.Washington.cw.net
                    (204.70.1.221) 209.496 ms  209.247 ms
 8  bordercore2.Boston.cw.net (166.48.64.1)
                    209.486 ms  209.332 ms  209.598 ms
 9  hill-associatesinc-internet.Boston.cw.net (166.48.67.54)
                    229.602 ms  219.510 ms *
10  syrup.hill.corn (208.162.106.3) 239.744 ms 239.348 m 219.607 ms
bsd: $
```

Рис. 4.10. Выдача traceroute с повторяющимися узлами

На рис. 4.10 показано еще одно интересное явление. Вы видите, что в узле 7 маршрут изменился после первой попытки. Возможно, это было вызвано тем, что маршрутизатор в узле 6 выполнил какие-то действия по балансированию нагрузки. А возможно, что узел `cpe3-fddi-0.washington.cw.net` за время, прошедшее с момента первой попытки, успел «отключиться», и вместо него был использован маршрутизатор с адресом `core6-serial5-1-0.Washington.cw.net`.

Еще одна проблема, встречающаяся, к сожалению, все чаще, состоит в том, что маршрутизаторы полностью блокируют все ICMP-сообщения. Некоторые организации, ошибочно полагая, что ICMP-сообщения несут какую-то опасность, отключают их. В таких условиях `traceroute` становится бесполезной, поскольку первый же такой узел, встретившийся на маршруте к получателю, с точки зрения `traceroute` ведет себя как «черная дыра». Никакая информация от последующих узлов не доходит, так как этот маршрутизатор отбрасывает и сообщение «истекло время в пути», и сообщение «порт недоступен».

Следующая проблема при работе с `traceroute` – это асимметрия маршрутов. Запуская `traceroute`, вы получаете маршрут от пункта отправления до пункта

назначения, но нет гарантии, что датаграмма, отправленная из пункта назначения, будет следовать тем же маршрутом. Хотя кажется естественным предположение о том, что почти все маршруты одинаковы, в действительности, как показано в работе [Paxson 1997], 49% изученных маршрутов демонстрируют асимметрию хотя бы в одном промежуточном узле.

Примечание *С помощью опции `-s`, которая устанавливает режим свободной маршрутизации, заданной источником (loose source routing) от пункта назначения в пункт отправления, теоретически можно получить оба маршрута. Но, как отмечает Джекобсон в комментариях к исходному тексту `traceroute`, количество маршрутизаторов, которые некорректно выполняют маршрутизацию, заданную источником, настолько велико, что этот метод на практике не работает. В главе 8 книги [Stevens 1994] объясняется суть метода и приводится пример его успешного применения.*

В другой работе Паксон отмечает, что асимметричные маршруты возникают также из-за эффекта «горячей картофелины» [Paxson 1995].

Примечание *Этот эффект состоит в следующем. Предположим, что хост А, расположенный на восточном побережье Соединенных Штатов, отправляет датаграмму хосту В на западном побережье. Хост А подключен к Internet через провайдера 1, а хост В – через провайдера 2. Допустим, что у обоих провайдеров есть опорные сети, проходящие через всю страну. Поскольку полоса пропускания опорной сети – это дефицитный ресурс, провайдер 1 пытается доставить датаграмму хосту в сети провайдера 2, пользуясь его же опорной сетью. Но точно так же, когда хост В отвечает, провайдер 2 пытается доставить ответ на противоположное побережье, пользуясь опорной сетью провайдера 1. Отсюда и асимметрия.*

Программа tracert в системе Windows

До сих пор описывалась UNIX-версия программы `traceroute`. Очень похожее средство – `tracert` – есть и в различных версиях операционной системы Windows. Программа `tracert` работает аналогично `traceroute`, но для определения маршрута используются не UDP-датаграммы, а эхо-запросы протокола ICMP (как в программе `ping`). В результате хост-получатель возвращает эхо-ответ ICMP, а не сообщение о недоступности порта. Промежуточные маршрутизаторы по-прежнему возвращают сообщение «истекло время в пути».

Примечание *В последних версиях `traceroute` есть опция `-I`, имитирующая такое же поведение. Подобную версию можно получить на сайте ftp://ftp.ee.lbl.gov/traceroute.tar.Z.*

Наверное, это изменение сделано исходя из соображения о том, что UDP-датаграммы часто отфильтровываются маршрутизаторами, тогда как эхо-запросы

и эхо-ответы ICMP, используемые программой ping, менее подвержены этому. Исходная версия traceroute также применяла эхо-запросы для определения маршрута, но потом они были заменены UDP-датаграммами, поскольку многие маршрутизаторы строго следовали предписанию RFC 792 [Postel 1981], требующему не посылать ICMP-сообщения в ответ на ICMP-сообщения [Jacobson 1999]. Действующее ныне RFC 1122 [Braden 1989] указывает, что ICMP-сообщение не должно посылаться в ответ на ICMP-сообщение *об ошибке*, но tracert по-прежнему встречает трудности в старых моделях маршрутизаторов.

В RFC 1393 [Malkin 1993] предложено добавить новую опцию в протокол IP и отдельное ICMP-сообщение, чтобы гарантировать надежность traceroute (а заодно и решить некоторые другие задачи), но, так как в маршрутизаторы и программное обеспечение хостов пришлось бы вносить изменения, этот метод не получил распространения.

Резюме

Утилита traceroute – очень полезный инструмент для диагностики сетевых ошибок, изучения маршрутизации и исследования топологии сети. Топология Internet нередко достаточно запутанна, и это может быть причиной неожиданного поведения приложений. С помощью traceroute зачастую удается обнаружить аномалии в сети, из-за которых программа ведет себя странно.

Программы traceroute и tracert работают путем отправки хосту назначения датаграммы с последовательно увеличивающимся значением в поле TTL. Затем они отслеживают приходящие от промежуточных маршрутизаторов ICMP-сообщения «истекло время в пути». Разница в том, что traceroute посылает UDP-датаграммы, а tracert – эхо-запросы ICMP.

Совет 36. Используйте программу ttcp

Часто необходимо иметь утилиту, которая может посылать произвольный объем данных другой (или той же самой) машине по протоколу TCP или UDP и собирать статистическую информацию о полученных результатах. В этой книге уже написано несколько программ такого рода. В этом разделе вы познакомитесь с готовым инструментом, обладающим той же функциональностью. Подобное средство можно использовать для тестирования собственного приложения или для получения информации о производительности конкретного стека TCP/IP или сети. Такая информация может оказаться бесценной на этапе создания прототипа.

Этот инструмент – программа ttcp, бесплатно распространяемая Лабораторией баллистических исследований армии США (BRL – Ballistics Research Laboratory). Ее авторы Майк Муусс (автор программы ping) и Терри Слэттери. Эта утилита доступна на множестве сайтов в Internet. В книге будет использована версия, которую Джон Лин модифицировал с целью включения дополнительной статистики; ее можно получить по анонимному FTP с сайта gwen.cs.purdue.edu из каталога /pub/lin. Версия без модификаций Лина находится, например, на сайте ftp.sgi.com в каталоге sgi/src/ttcp, в состав ее дистрибутива входит также страница руководства.

У программы ttcp есть несколько опций, позволяющих управлять: объемом посылаемых данных, длиной отдельных операций записи и считывания, размерами

буферов приема и передачи сокета, включением или отключением алгоритма Нейгла и даже выравниванием буферов в памяти. На рис. 4.11 приведена информация о порядке использования `ttcp`. Дается перевод на русский язык, хотя оригинальная программа, естественно, выводит справку по-английски.

```
Порядок вызова:ttcp -t [-опции] хост [ < in ]
               ttcp -r [-опции > out]
Часто используемые опции:
   -l ## длина в байтах буферов, в которые происходит считывание
      из сети и запись в сеть (по умолчанию 8192)
   -u использовать UDP, а не TCP
   -p ## номер порта, в который надо посылать данные или
      прослушивать (по умолчанию 5001)
   -s -t: отправить данные в сеть
      -r: считать (и отбросить) все данные из сети
   -A выравнивать начало каждого буфера на эту границу
      (по умолчанию 16384)
   -O считать, что буфер начинается с этого смещения
      относительно границы (по умолчанию 0)
   -v печатать более подробную статистику
   -d установить опцию сокета SO_DEBUG
   -b ## установить размер буфера сокета (если поддерживает
      операционная система)
   -f X формат для вычисления скорости обмена: k,K = кило(бит,байт);
                    m,M = мега; g,G = гига
Опции, употребляемые вместе с -t:
   -n ## число буферов, записываемых в сеть (по умолчанию 2048)
   -D не буферизовать запись по протоколу TCP (установить
      опцию сокета TCP_NODELAY)
Опции, употребляемые вместе с -r:
   -B для -s, выводить только полные блоки в соответствии
      с опцией -l (для TAR)
   -T "touch": обращаться к каждому прочитанному байту
```

Рис. 4.11. Порядок вызова ttcp

Поэкспериментируем с размером буфера передачи сокета. Сначала прогоним тест с размером буфера, выбранным по умолчанию, чтобы получить точку отсчета. В одном окне запустим экземпляр `ttcp`-потребителя:

```
bsd: $ ttcp -rsv
```

а в другом – экземпляр, играющий роль источника:

```
bsd: $ ttcp -tsv bsd
ttcp-t:  buflen=8192, nbuf=2048, align=16384/0, port=5013 tcp -> bsd
ttcp-t:  socket
ttcp-t:  connect
ttcp-t:  16777216 bytes in 1.341030 real seconds
      = 12217.474628 KB/sec (95.449021 Mb/sec)
ttcp-t:  16777216 bytes in 0.00 CPU seconds
      = 16384000.000000 KB/cpu sec
```

```
ttcp-t:  2048 I/O calls, msec/call = 0.67, calls/sec = 1527.18
ttcp-t:  buffer address 0x8050000
bds: $
```

Как видите, `ttcp` дает информацию о производительности. Для передачи 16 Мб потребовалось около 1,3 с.

Примечание *Аналогичная статистика печатается принимающим процессом, но поскольку цифры, по существу, такие же, они здесь не приводятся.*

Также был выполнен мониторинг обмена с помощью `tcpdump`. Вот типичная строка выдачи:

```
13:05:44.084576 bsd.1061 > bsd.5013: . 1:1449(1448)
                ack 1win17376 <nop,nop,timestamp 11306 11306> (DF)
```

Из нее видно, что TCP посылает сегменты по 1448 байт.

Теперь следует установить размер буфера передачи равным 1448 байт, и повторить эксперимент. Приемник данных нужно оставить без изменения.

```
bsd: $ ttcp -tsvb 1448 bsd
ttcp-t:  socket
ttcp-t:  sndbuf
ttcp-t:  connect
ttcp-t:  buflen=8192, nbuf=2048, align=16384/0, port=5013,
         sockbufsize=1448 tcp -> bsd
ttcp-t:  16777216 bytes in 2457.246699 real seconds
         = 6.667625 KB/sec (0.052091 Mb/sec)
ttcp-t:  16777216 bytes in 0.00 CPU seconds
         = 16384000.000000 KB/cpu sec
ttcp-t:  2048 I/O calls, msec/call = 1228.62, calls/sec = 0.83
ttcp-t:  buffer address 0x8050000
bds: $
```

На этот раз передача заняла почти 41 мин. Следует отметить, что, хотя по часам для передачи потребовалось больше 40 мин, время, затраченное процессором, по-прежнему очень мало, даже не поддается измерению. Поэтому, что бы ни произошло, это не связано с загрузкой процессора.

Теперь посмотрим, что показывает `tcpdump`. На рис. 4.12 приведены четыре типичные строки:

```
16:03:57.168093 bsd.1187 > bsd.5013: P 8193:9641(1448)
         ack 1 win 17376 <nop,nop,timestamp 44802 44802> (DF)
16:03:57.368034 bsd.5013 > bsd.1187: . ack 9641 win 17376
         <nop,nop,timestamp 44802 44802> (DF)
16:03:57.368071 bsd.1187 > bsd.5013: P 9641:11089(1448)
         ack 1 win 17376 <nop,nop,timestamp 44802 44802> (DF)
16:03:57.568038 bsd.5013 > bsd.1187: . ack 11089 win 17376
         <nop,nop,timestamp 44802 44802> (DF)
```

Рис. 4.12. Типичная выдача tcpdump для запуска ttcp -tsvb 1448 bsd

Обратите внимание, что время между последовательными сегментами составляет почти 200 мс. Возникает подозрение, что тут замешано взаимодействие между алгоритмами Нейгла и отложенного подтверждения (совет 24). И действительно именно ACK задерживаются.

Эту гипотезу можно проверить, отключив алгоритм Нейгла с помощью опции -D. Повторим эксперимент:

```
bsd: $ ttcp -tsvDb 1448 bsd
ttcp-t:  buflen=8192, nbuf=2048, align=16384/0, port=5013,
         sockbufsize=1448 tcp -> bsd
ttcp-t:  socket
ttcp-t:  sndbuf
ttcp-t:  connect
ttcp-t:  nodelay
ttcp-t:  16777216 bytes in 2457.396882 real seconds
         = 6.667218 KB/sec (0.052088 Mb/sec)
ttcp-t:  16777216 bytes in 0.00 CPU seconds
         = 16384000.000000 KB/cpu sec
ttcp-t:  2048 I/O calls, msec/call = 1228.70, calls/sec = 0.83
ttcp-t:  buffer address 0x8050000
bds: $
```

Как ни странно, ничего не изменилось.

Примечание *Это пример того, как опасно делать поспешные заключения. Стоило немного подумать и стало бы ясно, что алгоритм Нейгла тут ни при чем, так как посылаются заполненные сегменты. В частности, этому служит самый первый тест, – чтобы определить величину MSS.*

В совете 39 будут рассмотрены средства трассировки системных вызовов. Тогда вы вернетесь к этому примеру и обнаружите, что выполняемая ttcp операция записи не возвращает управление в течение примерно 1,2 с. Косвенное указание на это видно и из выдачи ttcp, где каждый вызов операции ввода/вывода занимает приблизительно 1,228 мс. Но, как говорилось в совете 15, TCP обычно не блокирует операции записи, пока буфер передачи не окажется заполненным. Таким образом, становится понятно, что происходит. Когда ttcp записывает 8192 байта, ядро копирует первые 1448 байт в буфер сокета, после чего блокирует процесс, так как места в буфере больше нет. TCP посылает все эти байты в одном сегменте, но послать больше не может, так как в буфере ничего не осталось.

Примечание *Из рис. 4.12 видно, что дело обстоит именно так, поскольку в каждом отправленном сегменте задан флаг PSH, а стеки, берущие начало в системе BSD, устанавливают этот флаг только тогда, когда выполненная операция передачи опустошает буфер.*

Поскольку приемник данных ничего не посылает в ответ, запускается механизм отложенного подтверждения, из-за которого ACK не возвращается до истечения тайм-аута в 200 мс.

В первом тесте TCP мог продолжать посылать заполненные сегменты данных, поскольку буфер передачи был достаточно велик (16 Кб на машине bsd) для сохранения нескольких сегментов. Трассировка системных вызовов для этого теста показывает, что на операцию записи уходит около 0,3 мс.

Этот пример наглядно демонстрирует, как важно, чтобы буфер передачи отправителя был, по крайней мере, не меньше буфера приема получателя. Хотя получатель был готов принимать данные и дальше, но в выходном буфере отправителя задержался последний посланный сегмент. Забыть про него нельзя, пока не придет ACK, говорящий о том, что данные дошли до получателя. Поскольку размер одного сегмента значительно меньше, чем буфер приема (16 Кб), его получение не приводит к обновлению окна (совет 15). Поэтому ACK задерживается на 200 мс. Подробнее о размерах буферов рассказано в совете 32.

Однако смысл этого примера в том, чтобы показать, как можно использовать ttcp для проверки эффекта установки тех или иных параметров TCP-соединения. Вы также видели, как анализ информации, полученной от ttcp, tcpdump и программы трассировки системных вызовов, может объяснить работу TCP.

Следует упомянуть о том, как использовать программу ttcp для организации «сетевого конвейера» между хостами. Например, скопировать всю иерархию каталогов с хоста A на хост B. На хосте B вводите команду

```
ttcp -rB | tar -xpf -
```

на хосте A – команду

```
tar -cf - каталог | ttcp -t A
```

Можно распространить конвейер на несколько машин, если на промежуточных запустить команду

```
ttcp -r | ttcp -t следующий_узел
```

Резюме

В этом разделе показано, как пользоваться программой ttcp для экспериментирования с различными параметрами TCP-соединения. ttcp можно применять также в целях тестирования собственных приложений, предоставляя для них источник или приемник данных, работающий по протоколу TCP либо UDP. И, наконец, вы видели, как использовать ttcp для организации сетевого конвейера между двумя или более машинами.

Совет 37. Применяйте программу lsof

В сетевом (и не только) программировании часто необходимо определить, какой процесс открыл файл или сокет. Особенно это важно в сетевом окружении, поскольку, как было показано в совете 16, при завершении процесса, работавшего с сокетом, FIN не будет послан, если другой процесс держит этот сокет открытым.

Хотя ситуация, когда другой процесс держит сокет открытым, выглядит странно, но она часто возникает, особенно при работе в UNIX. Происходит вот что: один процесс принимает соединение и запускает другой процесс, который

будет работать с этим соединением (кстати, именно это и делает inetd – совет 17). Если процесс, принявший соединение, не закроет сокет после создания процесса – потомка, то счетчик ссылок на это сокет будет равен двум. Поэтому после того как потомок закроет сокет, соединение останется открытым, и FIN не будет послан.

Та же проблема может возникнуть и по другой причине. Предположим, что хост клиента, работавшего с созданным процессом, аварийно остановился, в результате чего потомок «завис». Такая ситуация обсуждалась в совете 10. Если процесс, принимающий соединения, завершит работу, то перезапустить его будет невозможно (если, конечно, не была задана опция сокета SO_REUSEADDR, – совет 23), так как локальный порт уже привязан к созданному процессу.

В этих и некоторых других случаях необходимо знать, какой процесс (или процессы) держит сокет открытым. Утилита netstat (совет 38) сообщает, что некоторый процесс занимает данный порт или адрес, но что это за процесс, неизвестно. В некоторых версиях UNIX для ответа на этот вопрос есть программа fstat. Виктор Абель (Victor Abell) написал свободно распространяемую программу lsof, работающую почти во всех версиях UNIX.

Примечание *Дистрибутив lsof можно получить по анонимному FTP с сайта vic.cc.purdue.edu из каталога pub/tools/unix/lsof.*

lsof – это исключительно гибкая программа; руководство по ней занимает 26 печатных страниц. С ее помощью можно получить самую разнообразную информацию об открытых файлах. Как и в случае tcpdump, предоставление единого интерфейса к нескольким диалектам UNIX – это существенное достоинство.

Рассмотрим некоторые возможности lsof, полезные в сетевом программировании. В руководстве приводится подробная информация и о других ее применениях.

Предположим, что после выполнения команды netstat -af inet (совет 38) вы обнаруживаете, что некоторый процесс прослушивает порт 6000:

```
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address    Foreign Address   (state)
tcp        0      0  *.6000           *.*                LISTEN
```

Порт 6000 не относится к хорошо известным (совет 18), поэтому возникает вопрос: что же его прослушивает? Как уже упоминалось, в netstat по этому поводу ничего не говорится – она лишь сообщает о наличии прослушивающего процесса. Зато программа lsof не испытывает никаких затруднений:

```
bsd# lsof -i TCP:6000
COMMAND   PID USER FD  TYPE    DEVICE SIZE/OFF  NODE  NAME
XF86_Mach 253 root  0u inet0xf5d98840      0t0   TCP   *:6000  (LISTEN)
bsd#
```

Следует отметить, что вы запускали lsof от имени пользователя root. Это необходимо, потому что используемая версия lsof сконфигурирована для перечисления файлов, принадлежащих только данному пользователю, за исключением ситуации, когда ее запускает root. Это свойство направлено на обеспечение безопасности, но его можно отключить во время компиляции программы. Далее надо отметить,

что процесс был запущен пользователем root с помощью команды XF86_Mach. Это ваш X-сервер.

Опция -i TCP:6000 означает, что lsof должна искать открытые TCP-сокеты, привязанные к порту 6000. Можно было бы показать все TCP-сокеты с помощью опции -i TCP или все TCP- и UDP-сокеты – с помощью опции -i.

Предположим, что вы еще раз запустили netstat и обнаружили, что кто-то открыл FTP-соединение с хостом vic.cc.purdue.edu:

```
Active Internet connections
Proto Recv-Q Send-Q  Local Address    Foreign Address   (state)
tcp        0      0  bsd.1124        vic.cc.purdue.edu.ftp ESTABLISHED
```

Выяснить, кто это сделал, поможет lsof:

```
bsd# lsof -i @vic.cc.purdue.edu
COMMAND PID USER FD  TYPE     DEVICE SIZE/OFF NODE NAME
ftp     450 jcs   3u inet 0xf5d99f00      0t0  TCP bsd:1124->
   vic.cc.purdue.edu:ftp ESTABLISHED
bsd#
```

Как обычно, в имени машины bsd опущен домен и строка разбита на две. Из полученной выдачи видно, что FTP-соединение открыл пользователь jcs.

Необходимо подчеркнуть, что lsof может выдать информацию только об открытых файлах. Собственно говоря, название программы – аббревиатура list open files (перечислить открытые файлы). Это, в частности, означает, что с ее помощью нельзя получить информацию о TCP-соединениях, находящихся в состоянии TIME-WAIT (совет 22), поскольку с ними не связан никакой открытый сокет или файл.

Резюме

Здесь показано, как можно воспользоваться утилитой lsof для получения ответа на разнообразные вопросы об открытых файлах. К сожалению, нет версии lsof для Windows.

Совет 38. Используйте программу netstat

Ядро операционной системы ведет разнообразную статистику об объектах, имеющих отношение к сети. Эту информацию можно получить с помощью программы netstat. Существует четыре вида запросов.

Активные сокеты

Во-первых, можно получить сведения об активных сокетах. Хотя netstat дает информацию о разных типах сокетов, интерес представляют только сокеты из адресных доменов inet (AF_INET) и UNIX (AF_LOCAL или AF_UNIX). Можно потребовать вывести все типы сокетов или выбрать один тип, указав адресное семейство с помощью опции -f.

По умолчанию серверы, сокеты которых привязаны к адресу INADDR_ANY, не выводятся, но этот режим можно отключить с помощью опции -a. Например, если нужны TCP/UDP-сокеты, то можно вызвать netstat так:

```
bsd: $ netstat -f inet
Active Internet connections
Proto Recv-Q Send-Q  Local Address      Foreign Address  (state)
tcp        0      0  localhost.domain   *.*              LISTEN
tcp        0      0  bsd.domain         *.*              LISTEN
udp        0      0  localhost.domain   *.*
udp        0      0  bsd.domain         *.*
bsd: $
```

Здесь показан только сервер доменных имен (named), работающий на машине bsd. Если же нужно вывести все серверы, то программа запускается таким образом:

```
bsd: $ netstat -af inet
Active Internet connections
Proto Recv-Q Send-Q  Local Address     Foreign Address  (state)
tcp        0      0  *.6000              *.*            LISTEN
tcp        0      0  *.smtp              *.*            LISTEN
tcp        0      0  *.printer           *.*            LISTEN
tcp        0      0  *.rlnum             *.*            LISTEN
tcp        0      0  *.tcpmux            *.*            LISTEN
tcp        0      0  *.chargen           *.*            LISTEN
tcp        0      0  *.discard           *.*            LISTEN
tcp        0      0  *.echo              *.*            LISTEN
tcp        0      0  *.time              *.*            LISTEN
tcp        0      0  *.daytime           *.*            LISTEN
tcp        0      0  *.finger            *.*            LISTEN
tcp        0      0  *.login             *.*            LISTEN
tcp        0      0  *.cmd               *.*            LISTEN
tcp        0      0  *.telnet            *.*            LISTEN
tcp        0      0  *.ftp               *.*            LISTEN
tcp        0      0  *.1022              *.*            LISTEN
tcp        0      0  *.2049              *.*            LISTEN
tcp        0      0  *.1023              *.*            LISTEN
tcp        0      0  localhost.domain    *.*            LISTEN
tcp        0      0  bsd.domain          *.*            LISTEN
udp        0      0  *.udpecho           *.*
udp        0      0  *.chargen           *.*
udp        0      0  *.discard           *.*
udp        0      0  *.echo              *.*
udp        0      0  *.time              *.*
udp        0      0  *.ntalk             *.*
udp        0      0  *.biff              *.*
udp        0      0  *.1011              *.*
udp        0      0  *.nfsd              *.*
udp        0      0  *.1023              *.*
udp        0      0  *.sunrpc            *.*
udp        0      0  *.1024              *.*
udp        0      0  localhost.domain    *.*
udp        0      0  bsd.domain          *.*
udp        0      0  *.syslog            *.*
bsd: $
```

Если бы вы запустили программу `lsof` (совет 37), то обнаружили, что большинство этих «серверов» – в действительности `inetd` (совет 17), ожидающий прихода соединений или датаграмм в порты стандартных сервисов. Слово «`LISTEN`» в колонке `state` для TCP-соединений означает, что сервер ждет запроса на соединение от клиента.

Если обратиться к серверу эхо-контроля с помощью `telnet`:

```
bsd: $ telnet bsd echo
```

то появится соединение в состоянии `ESTABLISHED`:

```
Proto Recv-Q Send-Q  Local Address  Foreign Address   (state)
tcp        0      0  bsd.echo        bsd.1035         ESTABLISHED
tcp        0      0  bsd.1035        bsd.echo         ESTABLISHED
tcp        0      0  *.echo          *.*              LISTEN
```

Здесь опущены строки, не относящиеся к серверу эхо-контроля. Обратите внимание, что, поскольку вы соединились с локальной машиной, в выдаче `netstat` соединение присутствует дважды: один раз для клиента, а другой – для сервера. Заметьте также, что `inetd` продолжает прослушивать порт в ожидании дальнейших соединений.

Примечание *Последнее замечание требует еще нескольких пояснений. Хотя telnet-клиент подсоединился к порту 7 (порт эхо) и фактически использует его в качестве порта назначения, хост продолжает прослушивать этот порт. Это нормально, так как с точки зрения TCP соединение – это четверка, состоящая из локальных IP-адреса и порта и удаленных IP-адреса и порта (совет 23). Как видите,* `inetd` *прослушивает порт на универсальном «псевдоадресе»* `INADDR_ANY`*, что показано звездочкой в колонке Local Address, тогда как IP-адрес для установленного соединения равен* `bsd`*. Если бы вы создали одно дополнительное соединение с помощью telnet, то получили бы еще две строки, аналогичные первым двум, только порт клиента был бы отличен от 1035.*

Завершите работу клиента и снова запустите `netstat`. Вот что вы получите:

```
Proto Recv-Q Send-Q  Local Address     Foreign Address  (state)
tcp   0      0       bsd.1035          bsd.echo         TIME_WAIT
```

Как видно, клиентская сторона соединения находится в состоянии TIME-WAIT (совет 22). В колонке `state` могут появляться и другие состояния, подробнее о них рассказывается в RFC 793 [Postel 1981b].

Интерфейсы

С помощью `netstat` можно также получить информацию об интерфейсах. Такой пример был приведен в совете 7. Основная информация выдается при наличии опции `-i`:

```
bsd: $ netstat -i
Name  Mtu   Network       Address            Ipkts   Ierrs   Opkts   Oerrs   Coll
ed0   1500  <Link>        00.00.c0.54.53.73  40841   0       5793    0       0
ed0   1500  172.30        bsd                40841   0       5793    0       0
tun0* 1500  <Link>                           397     0       451     0       0
tun0* 1500  205.184.142 205.184.142.171      397     0       451     0       0
sl0*  552   <Link>                           0       0       0       0       0
lo0   16384 <Link>                           353     0       353     0       0
lo0   16384 127           localhost          353     0       353     0       0
```

Отсюда видно, что в машине bsd сконфигурировано четыре интерфейса. Первый – ed0– это адаптер сети Ethernet. Он входит в частную (RFC 1918 [Rekhter, Moskowitz et al. 1996]) сеть 172.30.0.0. Адрес 00.00.c0.54.73 – это первый в списке MAC-адресов (media access control – контроль доступа к носителю) данной сетевой карты. Через этот интерфейс прошло 40841 входных пакетов и 5793 выходных; не было зарегистрировано ни ошибок, ни коллизий. MTU (совет 7) составляет 1500 байт – максимальное значение для сетей Ethernet.

Интерфейс tun0 – это телефонный канал, по которому связь осуществляется по протоколу PPP (Point-to-Point Protocol). Он входит в сеть 205.184.142.0. MTU для этого интерфейса также составляет 1500 байт.

Интерфейс sl0 – это телефонный канал, по которому связь осуществляется по протоколу SLIP (Serial Line Internet Protocol), RFC 1055 [Romkey 1988]. Это еще один, ныне устаревший протокол двухточечного соединения по телефонным линиям. Данный интерфейс в машине bsd не используется.

Наконец, есть еще возвратный интерфейс lo0. О нем уже неоднократно говорилось.

В сочетании с опцией -i можно также задать опции -b или -d. Тогда будет напечатано количество байт, прошедших через интерфейс в обе стороны, или число отброшенных пакетов.

Маршрутная таблица

Кроме того, netstat может дать маршрутную таблицу. Назначьте опцию -n, чтобы получить не символические имена, а IP-адреса; так лучше видно, в какие сети маршрутизируются пакеты.

Интерфейсы и соединения, выведенные на рис. 4.13, показаны и на рис. 4.14. Интерфейс lo0 не показан, так как полностью находится внутри машины bsd.

```
bsd: $ netstat -rn
Routing tables

Internet:
Destination      Gateway            Flags  Refs  Use  Netif Expire
default          163.179.44.41      UGSc   2     0    tun0
127.0.0.1        127.0.0.1          UH     1     34   lo0
163.179.44.41    205.184.142.171    UH     3     0    tun0
172.30           link#1             UC     0     0    ed0
172.30.0.1       0:0:c0:54:53:73    UHLW   0     132  lo0
```

Рис. 4.13. Маршрутная таблица, выведенная программой netstat

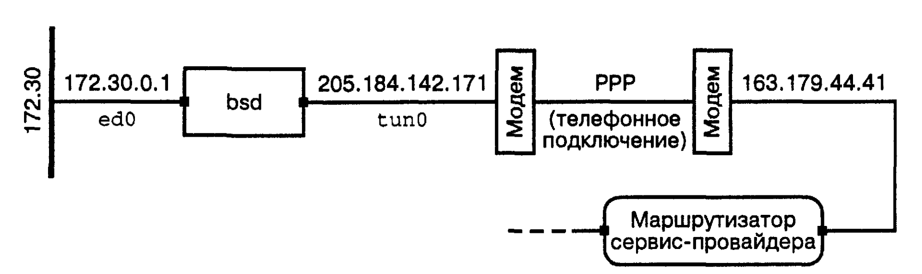


Рис. 4.14. Информация об интерфейсах и хостах, выведенная программой netstat

Прежде чем знакомиться с отдельными элементами этой выдачи, обсудим назначение колонок. В первой колонке находится пункт назначения маршрута. Это может быть конкретный хост, сеть или маршрут по умолчанию.

В колонке `Flags` печатаются различные флаги, большая часть которых зависит от реализации. Следует упомянуть только следующие:

- U – маршрут задействован («UP»);
- H – маршрут к хосту. Если этот флаг отсутствует, то речь идет о маршруте к сети (или к подсети, если используется бесклассовая междоменная маршрутизация CIDR – совет 2);
- G – непрямой маршрут. Иными словами, пункт назначения не связан напрямую с данным хостом, к нему следует добираться через промежуточный маршрутизатор или шлюз (G – gateway).

Легко сделать ошибку, полагая, что флаги H и G взаимоисключающие, то есть маршрут может идти либо к хосту (H), либо к промежуточному шлюзу (G). Флаг H означает, что адрес в первой колонке представляет собой полный IP-адрес хоста. Если флага H нет, то адрес в этой колонке не содержит идентификатора хоста, иными словами, он – адрес сети. Флаг G показывает, достижим ли адрес, проставленный в первой колонке, непосредственно с данного хоста или необходимо пройти через промежуточный маршрутизатор.

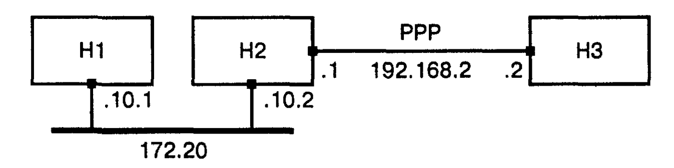


Рис. 4.15. H2 выступает в роли шлюза к H3

Вполне возможно, что для некоторого маршрута будут одновременно установлены флаги G и H. Рассмотрим, например, две сети, изображенные на рис. 4.15. Хосты H1 и H2 подключены к сети Ethernet с адресом 172.20. Хост H3 соединен с H2 по PPP-линии с сетевым адресом 198.168.2.

Маршрут к Н3 в маршрутной таблице Н1 будет выглядеть так:

```
Destination    Gateway       Flags    Refs    Use   Netif Expire
192.168.2.2    172.20.10.2   UGH      0       0     ed0
```

Флаг H установлен потому, что 192.168.2.2 – полный адрес хоста. А флаг G – так как Н1 не имеет прямого соединения с Н3 и должен идти через хост Н2 (172.20.10.2). Обратите внимание, что на рис. 4.13 для маршрута к хосту 163.179.44.41 нет флага G, поскольку этот хост напрямую подключен к интерфейсу tun0 (205.184.142.171) в машине bsd.

На рис. 2.9 в маршрутной таблице Н1 не должно быть записи для Н3. Вместо нее присутствует запись для подсети 190.50.2, поскольку именно в этом состоит смысл организации подсетей – уменьшить размеры маршрутных таблиц. Запись в маршрутной таблице Н1 для этой подсети выглядела бы так:

```
Destination    Gateway      Flags    Refs    Use    Netif  Expire
190.50.2       190.50.1.4   UG       0       0      ed0
```

Флаг H не установлен, так как 190.50.2 – адрес подсети, а не отдельного хоста. Имеется флаг G, так как Н3 не соединен напрямую с Н1. Датаграммы от Н3 к Н1 должны проходить через маршрутизатор R1 (190.50.1.4).

Смысл колонки Gateway зависит от того, есть флаг G или нет. Если маршрут непрямой (флаг G есть), то в колонке Gateway находится IP-адрес следующего узла (шлюза). Если же флага G нет, то в этой колонке печатается информация о том, как достичь напрямую подсоединенного пункта назначения. Во многих реализациях это всегда IP-адрес интерфейса, к которому и подсоединен пункт назначения. В реализациях, производных от BSD, это может быть также MAC-адрес, как показано в последней строке на рис. 4.13. В таком случае будет установлен флаг L.

Колонка Refs содержит счетчик ссылок на маршрут, то есть количество активных пользователей этого маршрута.

Колонка Use указывает, сколько пакетов было послано по этому маршруту, а колонка Netif содержит имя ассоциированного сетевого интерфейса, который представляет собой тот же объект, о котором вы получаете информацию с помощью опции -i.

Теперь, разобравшись, что означают колонки, печатаемые командой netstat -rn, вернемся к рис. 4.13.

Первая строка на этом рисунке описывает *маршрут по умолчанию*. Именно по нему отсылаются датаграммы, когда в маршрутной таблице нет более точного маршрута. Например, если выполнить команду ping netcom4.netcom.com, то получится такой результат:

```
bsd: $ ping netcom4.netcom.com
PING netcom4.netcom.com (199.183.9.104): 56 data bytes
64 bytes from 199.183.9.104: icmp_seq=0 ttl=248 time=268.604 ms
...
```

Поскольку нет маршрута ни до хоста 199.183.9.104, ни до сети, содержащей этот хост, эхо-запросы ICMP (совет 33) посылаются по маршруту по умолчанию. В соответствии с первой строкой выдачи netstat шлюз для этого маршрута имеет адрес 163.179.44.41, туда и посылается датаграмма. Строка 3 на рис. 4.13 показывает, что есть прямой маршрут к хосту 163.179.44.41, и отсылать ему датаграммы следует через интерфейс с IP-адресом 205.184.142.171.

Строка 2 в выдаче – это маршрут для возвратного адреса (127.0.0.1). Поскольку это адрес хоста, установлен флаг H. Так как хост подсоединен напрямую, то имеется и флаг G. А в колонке Gateway вы видите IP-адрес интерфейса lo0.

В строке 4 представлен маршрут к локальной сети Ethernet. В связи с тем, что на машине bsd установлена операционная система, производная от BSD, в колонке Gateway находится строка Link#1. В других системах был бы просто напечатан IP-адрес интерфейса, подсоединенного к локальной сети (172.30.0.1).

Статистика протоколов

С помощью netstat можно получить статистику протоколов. Если задать опцию -s, то netstat напечатает статистические данные по протоколам IP, ICMP, IGMP, UDP и TCP. Если нужен какой-то один протокол, то его можно указать посредством опции -p. Так, для получения статистики по протоколу UDP следует ввести следующую команду:

```
bsd: $ netstat -sp udp
udp:
   82 datagrams received
   0 with incomplete header
   0 with bad data length field
   0 with bad checksum
   1 dropped due to no socket
   0 broadcast/multicast datagrams dropped due to no socket
   0 dropped due to full socket buffers
   0 not for hashed pcb
   81 delivered
   82 datagrams output
bsd: $
```

Ниже дается перевод на русский язык, программа netstat использует английский.

```
udp:
   82 датаграмм получено
   0 с неполным заголовком
   0 с неправильным значением в поле длины данных
   0 с неправильной контрольной суммой
   1 отброшено из-за отсутствия сокета
   0 отброшено широковещательных/групповых датаграмм
     из-за отсутствия сокета
   0 отброшено из-за переполнения буфера сокета
   0 не для хэшированного блока управления протоколом
   81 доставлено
   82 отправлено датаграмм
```

Можно отменить печать строк с нулевыми значениями, если дважды задать опцию -s:

```
bsd: $ netstat -ssp udp
udp:
```

```
    82 datagrams received
    1 dropped due to no socket
    81 delivered
    82 datagrams output
bsd: $
```

Периодический просмотр статистики TCP оказывает очень «отрезвляющее» действие. На машине bsd netstat выводит для TCP 45 статистических показателей. Вот строки с ненулевыми значениями, которые были получены при запуске netstat-ssp tcp:

```
tcp:
    446 packets sent
        190 data packets (40474 bytes)
        213 ack-only packets (166 delayed)
        18 window update packets
        32 control packets
    405 packets received
        193 acks (for 40488 bytes)
        12 duplicate acks
        302 packets (211353 bytes) received in sequence
        10 completely duplicate packets (4380 bytes)
        22 out-of-order packets (16114 bytes)
        2 window update packets
    20 connection requests
    2 connection accepts
    13 connections established (including accepts)
    22 connection closed (including 0 drops)
        3 connections updated cached RTT on close
        3 connections updated cached RTT variance on close
    2 embryonic connections dropped
    193 segments updated rtt (of 201 attempts)
    31 correct ACK header predictions
    180 correct data packet header predictions
```

Далее дается перевод статистической информации на русский язык.

```
tcp:
    446 пакетов послано
        190 пакетов данных (40474 байта)
        213 пакетов, содержащих только ack (166 отложенных)
        18 пакетов с обновлением окна
        32 контрольных пакета
    405 пакетов принято
        193 ack (на 40488 байт)
        12 повторных ack
        302 пакета (211353 байта) получено по порядку
        10 пакетов - полных дубликатов (4380 байт)
        22 пакета не по порядку (16114 байта)
        2 пакета с обновлением окна
```

```
20 запросов на соединение
2 приема соединения
13 соединений установлено (включая принятые)
22 соединения закрыто (включая 0 сброшенных)
   3 соединения при закрытии обновили RTT в кэше
   3 соединения при закрытии обновили дисперсию RTT в кэше
2 эмбриональных соединения сброшено
193 сегмента обновили rtt (из 201 попыток)
31 правильное предсказание заголовка ACK
180 правильных предсказаний заголовка пакета с данными
```

Эта статистика получена после перезагрузки машины bsd и последовавших за ней отправки и получения нескольких сообщений по электронной почте, а также чтения нескольких телеконференций. Если предположить, что такие события, как доставка пакетов не по порядку или получение дубликатов пакетов, происходят очень редко, то полученная информация полностью развеет эти иллюзии. Так, из 405 полученных пакетов 10 оказались дубликатами, а 22 пришли не по порядку.

Примечание *В работе [Bennett et al. 1999] показано, что приход пакетов не по порядку не обязательно свидетельствует о неисправности. Также объясняется, почему в будущем следует ожидать широкого распространения этого явления.*

Программа netstat в Windows

Выше рассмотрено, как работает программа netstat в системе UNIX. В Windows тоже есть аналогичная программа, принимающая в основном те же опции и выдающая такие же данные. Формат выдачи очень напоминает то, что вы видели, хотя состав информации не такой полный.

Резюме

Здесь приведены утилита netstat и те сведения о системе, которые можно получить с ее помощью. netstat сообщает об активных сокетах, о сконфигурированных сетевых интерфейсах, о маршрутной таблице и о статистике протоколов. Иными словами, она выдает отчеты о самых разнообразных аспектах сетевой подсистемы, причем в различных форматах.

Совет 39. Применяйте средства трассировки системных вызовов

Иногда при отладке сетевых приложений нужно уметь трассировать обращения к ядру операционной системы. Вы уже встречались с подобной ситуацией в совете 36 и вскоре вернетесь к этому примеру.

В большинстве операционных систем есть разные способы трассировки системных вызовов. В BSD это утилита ktrace, в SVR4 (и Solaris) – truss, а в Linux – strace.

Все эти программы похожи, поэтому остановимся только на ktrace. Беглого знакомства с руководством по truss или strace должно быть достаточно для применения аналогичной методики в других системах.

Преждевременное завершение

Первый пример – это вариация на тему первой версии программы shutdownc (листинг 3.1), которая разработана в совете 16. Идея программ badclient и shutdownc та же: читаются данные из стандартного ввода, пока не будет получен признак конца файла. В этот момент вы вызываете shutdown для отправки FIN-сегмента удаленному хосту, а затем продолжаете читать от него данные, пока не получите EOF, что служит признаком прекращения передачи удаленным хостом. Текст программы badclient приведен в листинге 4.2.

Листинг 4.2. Некорректный эхо-клиент

badclient.c

```
#include "etcp.h"
int main( int argc, char **argv )
{
    SOCKET s;
    fd_set readmask;
    fd_set allreads;
    int rc;
    int len;
    char lin[ 1024 ];
    char lout[ 1024 ];
    INIT();
    s = tcp_client( argv[ optind ], argv[ optind + 1 ] );
    FD_ZERO( &allreads );
    FD_SET( 0, &allreads );
    FD_SET( s, &allreads );
    for ( ;; )
    {
      readmask = allreads;
      rc = select( s + 1, &readmask, NULL, NULL, NULL );
      if ( rc <= 0 )
         error( 1, errno, "select вернула (%d)", rc );
      if ( FD_ISSET( s, &readmask ) )
      {
         rc = recv( s, lin, sizeof( lin ) - 1, 0 );
         if ( rc < 0 )
            error( 1, errno, "ошибка вызова recv" );
         if ( rc == 0 )
            error( 1, 0, "сервер отсоединился\n" );
         lin[ rc ] = '\0';
         if ( fputs( lin, stdout ) )
            error( 1, errno, "ошибка вызова fputs" );
      }
```

```
33      if ( FD_ISSET( 0, &readmask ) )
34   {
35         if ( fgets( lout, sizeof( lout ), stdin ) == NULL )
36         {
37            if ( shutdown( s, 1 ) )
38               error( 1, errno, "ошибка вызова shutdown" );
39         }
40         else
41         {
42            len = strlen( lout );
43            rc = send( s, lout, len, 0 );
44            if ( rc < 0 )
45               error( 1, errno, "ошибка вызова send" );
46            }
47      }
48   }
49 }
```

badclient.c

22-32 Если select показывает, что произошло событие чтения на соединении, пытаемся читать данные. Если получен признак конца файла, то удаленный хост прекратил передачу, поэтому завершаем работу. В противном случае выводим только что прочитанные данные на stdout.

33-47 Если select показывает, что произошло событие чтения на стандартном вводе, вызываем fgets для чтения данных. Если fgets возвращает NULL, что является признаком ошибки или конца файла, то вызываем shutdown, чтобы сообщить удаленному хосту о прекращении передачи. В противном случае посылаем только что прочитанные данные.

А теперь посмотрим, что произойдет при запуске программы badclient. В качестве сервера в этом эксперименте будет использоваться программа tcpecho (листинг 3.2). Следует напомнить (совет 16), что вы можете задать число секунд, на которое tcpecho должна задержать отправку ответа на запрос. Установите задержку в 30 с. Запустив клиент, напечатайте **hello** и сразу нажмите **Ctrl+D**, таким образом посылается fgets признак конца файла.

```
bsd: $ tcpecho 9000 30
       спустя 30 с
tcpecho: ошибка вызова recv:
  Connection reset by peer (54)
bsd: $
```

```
bsd: $ badclient bsd 9000
hello
^D
badclient: сервер отсоединился
bsd: $
```

Как видите, badclient завершает сеанс сразу же с сообщением о том, что сервер отсоединился. Но tcpecho продолжает работать и «спит», пока не истечет 30 с тайм-аута. После этого программа получает от своего партнера ошибку Connection reset by peer.

Это удивительно. Ожидалось, что tcpecho через 30 с пошлет эхо-ответ, а затем завершит сеанс, прочтя признак конца файла. Вместо этого badclient завершает работу немедленно, а tcpecho получает ошибку чтения.

Правильнее начать исследование проблемы с использования `tcpdump` (совет 34), чтобы понять, что же на самом деле посылают и принимают обе программы. Выдача `tcpdump` приведена на рис. 4.16. Здесь опущены строки, относящиеся к фазе установления соединения, и разбиты длинные строки.

```
1 18:39:48.535212 bsd.2027 > bsd.9000:
   P 1:7(6) ack 1 win 17376 <nop,nop,timestamp 742414 742400> (DF)
2 18:39:48.546773 bsd.9000 > bsd.2027:
   . ack 7 win 17376 <nop,nop,timestamp 742414 742414> (DF)
3 18:39:49.413285 bsd.2027 > bsd.9000:
   F 7:7(0) ack 1 win 17376 <nop,nop,timestamp 742415 742414> (DF)
4 18:39:49.413311 bsd.9000 > bsd.2027:
   . ack 8 win 17376 <nop,nop,timestamp 742415 742415> (DF)
5 18:40:18.537119 bsd.9000 > bsd.2027:
   P 1:7(6) ack 8 win 17376 <nop,nop,timestamp 742474 742415> (DF)
6 18:40:18.537180 bsd.2027 > bsd.9000:
   R 2059690956:2059690956(0) win 0
```

Рис. 4.16. Текст, выведенный tcpdump для программы badclient

Все выглядит нормально, кроме последней строки. Программа `badclient` посылает `tcpecho` строку `hello` (строка 1), а спустя секунду появляется сегмент FIN, посланный в результате `shutdown` (строка 3). Программа `tcpecho` в обоих случаях отвечает сегментом ACK (строки 2 и 4). Через 30 с после того, как `badclient` отправила `hello`, `tcpecho` отсылает эту строку назад (строка 5), но другая сторона вместо того, чтобы послать ACK, возвращает RST (строка 6), что и приводит к печати сообщения `Connection reset by peer`. RST был послан, поскольку программа `badclient` уже завершила сеанс.

Но все же видно, что `tcpecho` ничего не сделала для преждевременного завершения работы клиента, так что вся вина целиком лежит на `badclient`. Посмотрим, что же происходит внутри `badclient`, поможет в этом трассировка системных вызовов.

Повторим эксперимент, только на этот раз следует запустить программу так:

```
bsd: $ ktrace badclient bsd 9000
```

При этом `badclient` работает, как и раньше, но дополнительно вы получаете трассу выполняемых системных вызовов. По умолчанию трасса записывается в файл `ktrace.out`. Для печати содержимого этого файла надо воспользоваться программой `kdump`. Результаты показаны на рис. 4.17, в котором опущено несколько начальных вызовов, относящихся к запуску приложения и установлению соединения.

Первые два поля в каждой строке – это идентификатор процесса и имя исполняемой программы. В строке 1 вы видите вызов `read` с дескриптором `fd`, равным (`stdin`). В строке 2 читается шесть байт (GIO – сокращение от general I/O – общий ввод/вывод), содержащих `hello\n`. В строке 3 показано, что вызов `read` вернул 6 – число прочитанных байтов. Аналогично из строк 4–6 видно, что программа `badclient` писала в дескриптор 3, который соответствует сокету, соединенному с `tcpecho`. Далее, в строках 7 и 8 показан вызов `select`, вернувший единицу.

```
 1 4692 badclient CALL     read(0,0x804e000,0x10000)
 2 4692 badclient GIO fd   0 read 6 bytes
   "hello
   "
 3 4692 badclient RET      read 6
 4 4692 badclient CALL     sendto(0x3,0xefbfce68,0x6,0,0,0)
 5 4692 badclient GIO      fd 3 wrote 6 bytes
   "hello
   "
 6 4692 badclient RET      sendto 6
 7 4692 badclient CALL     select(0x4,0xefbfd6f0,0,0,0)
 8 4692 badclient RET      select 1
 9 4692 badclient CALL     read(0,0x804e000,0x10000)
10 4692 badclient GIO fd 0 read 0 bytes
   ""
11 4692 badclient RET      read 0
12 4692 badclient CALL     shutdown(0x3,0x1)
13 4692 badclient RET      shutdown 0
14 4692 badclient CALL     select(0x4,0xefbfd6f0,0,0,0)
15 4692 badclient RET      select 1
16 4692 badclient CALL     shutdown(0x3,0x1)
17 4692 badclient RET      shutdown 0
18 4692 badclient CALL     select(0x4,0xefbfd6f0,0,0,0)
19 4692 badclient RET      select 2
20 4692 badclient CALL     recvfrom(0x3,0xefbfd268,0x3ff,0,0,0)
21 4692 badclient GIO      fd 3 read 0 bytes
   ""
22 4692 badclient RET      recvfrom 0
23 4692 badclient CALL     write(0x2,0xefbfc6f4,0xb)
24 4692 badclient GIO      fd 2 wrote 11 bytes
   "badclient: "
25 4692 badclient RET      write 11/0xb
26 4692 badclient CALL     write(0x2,0xefbfc700,0x14)
27 4692 badclient GIO      fd 2 wrote 20 bytes
   "server disconnected
   "
28 4692 badclient RET      write 20/0x14
29 4692 badclient CALL     exit(0x1)
```

Рис. 4.17. Результаты прогона badclient под управлением ktrace

Это означает, что произошло одно событие. В строках 9–11 `badclient` прочитала EOF из `stdin` и вызвала `shutdown` (строки 12 и 13).

До сих пор все шло нормально, но вот в строках 14–17 вас поджидает сюрприз: `select` возвращает одиночное событие, и снова вызывается `shutdown`. Ознакомившись с листингом 4.2, вы видите, что такое возможно только при условии, если дескриптор `0` снова готов для чтения. Но `read` не вызывается, как можно было бы ожидать, ибо `fgets` в момент нажатия **Ctrl+D** отметила, что поток находится в конце файла, поэтому она возвращается, не выполняя чтения.

Примечание *Вы можете убедиться в этом, познакомившись с эталонной реализацией* fgets *(на основе* fgetc*) в книге [Kernighan and Ritchie 1988].*

В строках 18 и 19 select возвращает информацию о событиях на обоих дескрипторах stdin и сокете. В строках 20–22 видно, что recvfrom возвращает нуль (конец файла), а оставшаяся часть трассы показывает, как badclient выводит сообщение об ошибке и завершает сеанс.

Теперь ясно, что произошло: select показывает, что стандартный ввод готов для чтения в строке 15, поскольку вы забыли вызвать FD_CLR для stdin после первого обращения к shutdown. А следующий (уже второй) вызов shutdown вынуждает TCP закрыть соединение.

Примечание *В этом можно убедиться, посмотрев код на странице 1014 книги [Wright and Stevens 1995], где показано, что в результате обращения к* shutdown *вызывается функция* tcp_usrclosed*. Если* shutdown *уже вызывался раньше, то соединение находится в состоянии FIN-WAIT-2 и* tcp_usrclosed *вызывает функцию* soisdisconnected *(строка 444 на странице 1021). Этот вызов окончательно закрывает сокет и заставляет* select *вернуть событие чтения. А в результате будет прочитан EOF.*

Поскольку соединение закрыто, recvfrom возвращает нуль, то есть признак конца файла, и badclient выводит сообщение «сервер отсоединился» и завершает сеанс.

Ключ к пониманию событий в этом примере дал второй вызов shutdown. Легко обнаружилось отсутствующее обращение к FD_CLR.

Низкая производительность ttcp

Следующая ситуация – это продолжение примера из совета 36. Помните, что при размере буфера равном MSS соединения, время передачи 16 Мб возросло с 1,3 с до почти 41 мин.

На рис. 4.18 приведена репрезентативная выборка из результатов прогона ktrace для этого примера.

```
12512 ttcp  0.000023 CALL   write(0x3,0x8050000, 0x2000)
12512 ttcp  1.199605 GIO    fd 3 wrote 8192 bytes
    ""
12512 ttcp  0.000442 RET    write 8192/0x2000
12512 ttcp  0.000022 CALL   write(0x3,0x8050000, 0x2000)
12512 ttcp  1.199574 GIO    fd 3 wrote 8192 bytes
    ""
12512 ttcp  0.000442 RET    write 8192/0x2000
12512 ttcp  0.000023 CALL   write(0x3,0x8050000, 0x2000)
12512 ttcp  1.199514 GIO    fd 3 wrote 8192 bytes
    ""
12512 ttcp  0.000432 RET    write 8192/0x2000
```

Рис. 4.18. Выборка из результатов проверки ttcp -tsvb 1448 bsd под управлением ktrace

Вызвана kdump со следующими опциями:

```
kdump -R -m -l
```

для печати интервалов времени между вызовами и запрета вывода 8 Кб данных, ассоциированных с каждым системным вызовом.

Время каждой операции записи колеблется около значения 1,2 с. На рис. 4.19 для сравнения приведены результаты эталонного теста. На этот раз разброс значений несколько больше, но среднее время записи составляет менее 0,5 мс.

Большее время в записях типа GIO на рис. 4.18 по сравнению с временем на рис. 4.19 наводит на мысль, что операции записи блокировались в ядре (совет 36). Тогда становится понятна истинная причина столь резкого увеличения времени передачи.

```
12601 ttcp  0.000033 CALL  write(0x3,0x8050000, 0x2000)
12601 ttcp  0.000279 GIO   fd 3 wrote 8192 bytes
   ""
12601 ttcp  0.000360 RET   write 8192/0x2000
12601 ttcp  0.000033 CALL  write(0x3,0x8050000, 0x2000)
12601 ttcp  0.000527 GIO   fd 3 wrote 8192 bytes
   ""
12601 ttcp  0.000499 RET   write 8192/0x2000
12601 ttcp  0.000032 CALL  write(0x3,0x8050000, 0x2000)
12601 ttcp  0.000282 GIO   fd 3 wrote 8192 bytes
   ""
12601 ttcp  0.000403 RET   write 8192/0x2000
```

Рис. 4.19. Репрезентативная выборка из результатов проверки ttcp -tsv bsd под управлением ktrace

Резюме

Здесь описано два способа применения утилиты трассировки системных вызовов. В первом примере ошибку удалось обнаружить путем анализа системных вызовов, выполненных приложением. Во втором примере надо было отслеживать не очередность системных вызовов, а время выполнения некоторых из них.

Ранее уже говорилось о том, что для выяснения причин аномального поведения программы часто бывает необходимо сопоставить результаты, полученные от различных утилит. Программы трассировки системных вызовов, такие как ktrace, truss и strace, – это еще одно средство анализа в арсенале сетевого программиста.

Совет 40. Создание и применение программы для анализа ICMP-сообщений

Иногда необходимо знать, какие сообщения приходят в протоколе ICMP. Конечно, для их перехвата всегда можно воспользоваться программой tcpdump или другим сетевым анализатором, но иногда простой инструмент оказывается более удобным. Применение tcpdump влечет за собой некоторое снижение производительности, а также угрозу безопасности, хотя прослушивание ICMP-сообщений совершенно безобидно и ненакладно.

Во-первых, для работы такового сетевого анализатора, как tcpdump, нужно перевести сетевой интерфейс в режим пропускания. Это увеличивает нагрузку на центральный процессор, так как прерывание будет возникать при проходе через интерфейс каждого пакета Ethernet, даже если он адресован не той машине, на которой работает анализатор.

Во-вторых, во многих организациях применение сетевых анализаторов ограничено или вообще запрещено из-за потенциальной опасности перехвата информации и кражи паролей. Поэтому чтение ICMP-сообщений там более приемлемо.

В данном разделе разработан инструмент, который позволяет отслеживать ICMP-сообщения и не имеет недостатков, присущих сетевому анализатору общего назначения. Это позволит изучить простые сокеты, с которыми вы пока не сталкивались.

В совете 33 упоминалось, что ICMP-сообщения транспортируются в составе IP-датаграмм. Обычно содержимое ICMP-сообщения зависит от его типа, но интерес представляют только поля icmp_type и icmp_code, показанные на рис. 4.20. Дополнительные поля будут рассмотрены в связи с сообщениями о недоступности ресурса.

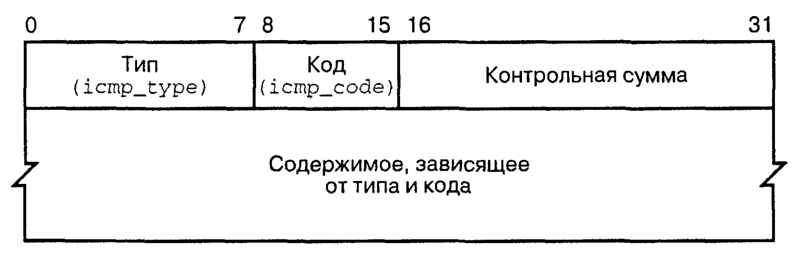


Рис. 4.20. Общий формат ICMP-сообщения

Часто возникают недоразумения при ответе на вопрос, что такое простые сокеты и для чего они нужны. Простые сокеты нельзя использовать для перехвата TCP-сегментов или UDP-датаграмм, поскольку они таким сокетам не передаются. Не годятся они и для получения всех ICMP-сообщений. Например, в системах, производных от BSD, эхо-запросы ICMP, запросы о временном штампе и запросы маски адреса полностью обрабатываются ядром и не передаются простым сокетам. В общем случае простой сокет получает все IP-датаграммы, в заголовках которых указан неизвестный ядру протокол, большинство ICMP-сообщений и все без исключения IGMP-сообщения.

Важно также отметить, что в простой сокет поступает вся IP-датаграмма *целиком*, включая заголовок. Ваша программа должна будет пропускать IP-заголовок.

Чтение ICMP-сообщений

Начнем с включаемых в программу файлов и функции main (листинг 4.3).

Листинг 4.3. Функция main программы icmp

icmp.c

```
#include <sys/types.h>
#include <netinet/in_systm.h>
```

```
 3 #include <netinet/in.h>
 4 #include <netinet/ip.h>
 5 #include <netinet/ip_icmp.h>
 6 #include <netinet/udp.h>
 7 #include "etcp.h"
 8 int main( int argc, char **argv )
 9 {
10     SOCKET s;
11     struct protoent *pp;
12     int rc;
13     char icmpdg[ 1024 ];
14     INIT();
15     pp = getprotobyname( "icmp" );
16     if ( pp == NULL )
17        error( 1, errno, "ошибка вызова getprotobyname" );
18     s = socket( AF_INET, SOCK_RAW, pp->p_proto );
19     if ( !isvalidsock( s ) )
20        error( 1, errno, "ошибка вызова socket" );
21     for ( ;; )
22     {
23        rc = recvfrom( s, icmpdg, sizeof( icmpdg ), 0,
24          NULL, NULL );
25        if ( rc < 0 )
26          error( 1, errno, "ошибка вызова recvfrom" );
27        print_dg( icmpdg, rc );
28     }
29 }
```

icmp.c

Открытие простого сокета

15-20 Поскольку использован простой сокет, надо указать нужный прото-кол. Вызов функции `getprotobyname` возвращает структуру, содержащую номер протокола ICMP. Обратите внимание, что в качестве типа указана константа `SOCK_RAW`, а не `SOCK_STREAM` или `SOCK_DGRAM`, как раньше.

Цикл обработки событий

21-28 Читаем каждую IP-датаграмму, используя `recvfrom`, как и в случае UDP-датаграмм. Для печати поступающих ICMP-сообщений вызываем функцию `print_dg`.

Печать ICMP-сообщений

Далее рассмотрим форматирование и печать ICMP-сообщений. Это делает функция `print_dg`, показанная в листинге 4.4. Передаваемый этой функции буфер имеет структуру, показанную на рис. 4.21.

Из рис. 4.21 видно, что буфер содержит IP-заголовок, за которым идет собственно ICMP-сообщение.

0–3	4–7	8–15	16–31			
Версия (ip_v)	Длина заголовка (ip_hl)	Тип сервиса	Полная длина (в байтах)			
Идентификация			0	DF	MF	Смещение фрагмента
Время существования		Протокол	Контрольная сумма IP-заголовка			
Адрес источника (ip_src)						
Адрес назначения (ip_dst)						
Опции IP (если есть)						
Тип (icmp_type)		Код (icmp_code)	Контрольная сумма ICMP-сообщения			
Дополнительные ICMP-данные						

IP-заголовок — ICMP

DF = флаг «не фрагментировать»
MF = флаг «есть еще фрагменты»

Рис. 4.21. ICMP-сообщение, передаваемое функции print_dg

Листинг 4.4. Функция print_dg

icmp.c

```
static void print_dg( char *dg, int len )
{
   struct ip *ip;
   struct icmp *icmp;
   struct hostent *hp;
   char *hname;
   int hl;
   static char *redirect_code[] =
   {
      "сеть", "хост",
      "тип сервиса и сеть", "тип сервиса и хост"
   };
   static char *timexceed_code[] =
   {
      "транзите", "сборке"
   };
   static char *param_code[] =
   {
```

```
19          "Плохой IP-заголовок", "Нет обязательной опции"
20      };
21      ip = ( struct ip * )dg;
22      if ( ip->ip_v != 4 )
23      {
24          error( 0, 0, "IP-датаграмма не версии 4\n" );
25          return;
26      }
27      hl = ip->ip_hl << 2;  /* Длина IP-заголовка в байтах. */
28      if ( len < hl + ICMP_MINLEN )
29      {
30          error( 0, 0, "short datagram (%d bytes) from %s\n",
31              len, inet_ntoa( ip->ip_src ) );
32          return;
33      }
34      hp = gethostbyaddr( ( char * )&ip->ip_src, 4, AF_INET );
35      if ( hp == NULL )
36          hname = "";
37      else
38          hname = hp->h_name;
39      icmp = ( struct icmp * )( dg + hl );  /* ICMP-пакет. */
40      printf( "ICMP %s (%d) от %s (%s)\n",
41          get_type( icmp->icmp_type ),
42          icmp->icmp_type, hname, inet_ntoa( ip->ip_src ) );
43      if ( icmp->icmp_type == ICMP_UNREACH )
44          print_unreachable( icmp );
45      else if ( icmp->icmp_type == ICMP_REDIRECT )
46          printf( "\tПеренаправление на %s\n", icmp->icmp_code <= 3 ?
47              redirect_code[ icmp->icmp_code ] : "Некорректный код" );
48      else if ( icmp->icmp_type == ICMP_TIMXCEED )
49          printf( "\tTTL == 0 при %s\n", icmp->icmp_code <= 1 ?
50          timexceed_code[ icmp->icmp_code ] : "Некорректный код" );
51      else if ( icmp->icmp_type == ICMP_PARAMPROB )
52          printf( "\t%s\n", icmp->icmp_code <= 1 ?
53              param_code[ icmp->icmp_code ] : "Некорректный код" );
54  }
```

icmp.c

Получение указателя на IP-заголовок и проверка корректности пакета

21 Записываем в переменную `ip` указатель на только что прочитанную датаграмму, приведенный к типу `struct ip *`.

22-26 Поле `ip_v` – это версия протокола IP. Если протокол не совпадает с IPv4, то печатаем сообщение об ошибке и выходим.

27-33 Поле `ip_hl` содержит длину заголовка в 32-байтных словах. Умножаем его на 4, чтобы получить длину в байтах, и сохраняем результат в переменной `hl`. Затем проверяем, что длина ICMP-сообщения не меньше минимально допустимой величины.

Получение имени хоста отправителя

34-38 Используем адрес источника в ICMP-сообщении, чтобы найти имя хоста отправителя. Если `gethostbyaddr` вернет `NULL`, то записываем в hname пустую строку, в обратном случае – имя хоста.

Пропуск IP-заголовка и печать отправителя и типа

39-42 Устанавливаем указатель `icmp` на первый байт, следующий за IP-заголовком. Этот указатель используется далее для получения типа ICMP-сообщения (`icmp_type`) и печати типа, адреса и имени хоста отправителя. Для получения ASCII-представления типа ICMP вызываем функцию `get_type`, текст которой приведен в листинге 4.5.

Печать информации, соответствующей типу

43-44 Если это одно из ICMP-сообщений о недоступности, то вызываем функцию `print_unreachable` (листинг 4.6) для печати дополнительной информации.

45-47 Если это сообщение о перенаправлении, то получаем тип перенаправления из поля `icmp_code` и печатаем его.

48-50 Если это сообщение об истечении времени существования, из поля `icmp_code` узнаем, произошло ли это во время транзита или сборки датаграммы, и печатаем результат.

51-53 Если это сообщение о некорректном параметре, из поля `icmp_code` определяем, в чем ошибка, и печатаем результат.

Функция `get_type` очевидна. Вы проверяете допустимость кода типа и возвращаете указатель на соответствующую строку (листинг 4.5).

Листинг 4.5. Функция get_type

icmp.c

```
static char *get_type( unsigned icmptype )
{
  static char *type[] =
  {
     "Эхо-ответ",                            /* 0 */
     "ICMP Тип 1",                           /* 1 */
     "ICMP Тип 2",                           /* 2 */
     "Пункт назначения недоступен",          /* 3 */
     "Источник приостановлен",               /* 4 */
     "Перенаправление",                      /* 5 */
     "ICMP Тип 6",                           /* 6 */
     "ICMP Тип 7",                           /* 7 */
     "Эхо-запрос",                           /* 8 */
     "Отклик маршрутизатора",                /* 9 */
     "Поиск маршрутизаторов",                /* 10 */
     "Истекло время существования",          /* 11 */
     "Неверный параметр",                    /* 12 */
     "Запрос временного штампа",             /* 13 */
     "Ответ на запрос временного штампа",    /* 14 */
     "Запрос информации",                    /* 15 */
```

```
    "Ответ на запрос информации",            /* 16 */
    "Запрос маски адреса",                    /* 17 */
    "Ответ на запрос маски адреса"           /* 18 */
};
if ( icmptype < ( sizeof( type ) / sizeof( type[ 0 ] ) ) )
    return type[ icmptype ];
return "НЕИЗВЕСТНЫЙ ТИП";
}
```

icmp.c

Последняя функция – это `print_unreachable`. ICMP-сообщения о недоступности содержат IP-заголовок и первые восемь байт той IP-датаграммы, из-за которой было сгенерировано сообщение о недоступности. Это позволяет узнать адреса и номера портов отправителя и предполагаемого получателя недоставленного сообщения.

Структура IP-датаграммы, прочитанной из простого сокета в составе ICMP-сообщения о недоступности, показана на рис. 4.22. Та часть, которую уже обработала функция `print_dg`, заштрихована, она не передается в `print_unreachable`. Приведены также входной параметр функции `print_unreachable` – `icmp` и локальные переменные `ip` и `udp`.

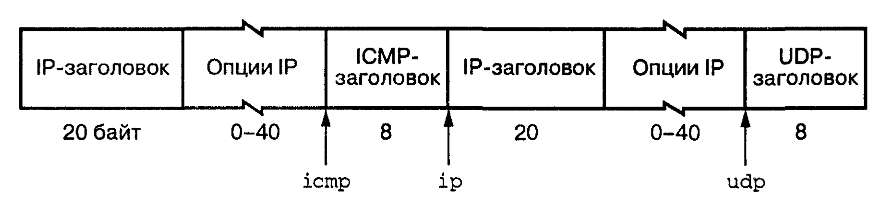


Рис. 4.22. ICMP-сообщение о недоступности

Функция `print_unreachable` извлекает информацию из заголовка и первых восьми байт включенной IP-датаграммы. Хотя вы пометили байты как UDP-заголовок, это мог быть и заголовок TCP: номера портов в обоих случаях находятся в одной и той же позиции. Формат UDP-заголовка показан на рис. 4.23.

0 — 15 | 16 — 31

Номер порта источника (`uh_sport`)	Номер порта назначения (`uh_dport`)
Длина UDP-датаграммы	Контрольная сумма

Рис. 4.23. UDP-заголовок

Текст функции `print_unreachable` приведен в листинге 4.6.

Листинг 4.6. Функция print_unreachable

icmp.c

```
static void print_unreachable( struct icmp *icmp )
{
```

```
3  struct ip *ip;
4  struct udphdr *udp;
5  char laddr[ 15 + 1 ];
6  static char *unreach[] =
7  {
8     "Сеть недоступна",                               /* 0 */
9     "Хост недоступен",                               /* 1 */
10    "Протокол недоступен",                           /* 2 */
11    "Порт недоступен",                               /* 3 */
12    "Нужна фрагментация, поднят бит DF",             /* 4 */
13    "Ошибка маршрутизации от источника",             /* 5 */
14    "Сеть назначения неизвестна",                    /* 6 */
15    "Хост назначения неизвестен",                    /* 7 */
16    "Хост источника изолирован",                     /* 8 */
17    "Сеть назначения закрыта администратором ",     /* 9 */
18    "Хост назначения закрыт администратором ",      /* 10 */
19    "Сеть недоступна для типа сервиса",              /* 11 */
20    "Хост недоступен для типа сервиса",              /* 12 */
21    "Связь запрещена администратором",               /* 13 */
22    "Нарушение предшествования хостов",              /* 14 */
23    "Действует отсечка предшествования"              /* 15 */
24 };
25 ip = ( struct ip * )( ( char * )icmp + 8 );
26 udp = ( struct udphdr * )( ( char * )ip + ( ip->ip_hl << 2 ) );
27 strcpy( laddr, inet_ntoa( ip->ip_src ) );
28 printf( "\t%s\n\tИст.: %s.%d, Назн.: %s.%d\n",
29    icmp->icmp_code < ( sizeof( unreach ) /
30    sizeof( unreach[ 0 ] ) )?
31       unreach[ icmp->icmp_code ] : "Некорректный код",
32    laddr, ntohs( udp->uh_sport ),
33    inet_ntoa( ip->ip_dst ), ntohs( udp->uh_dport ) );
34}
```

icmp.c

Установка указателей и получение адреса источника

25-26 Начинаем с установки указателей `ip` и `udp` соответственно на IP-заголовок и первые восемь байт вложенной IP-датаграммы.

27 Копируем адрес источника из IP-заголовка в локальную переменную `laddr`.

Печать адресов, портов и типа сообщения

28-33 Печатаем адреса и номера портов источника и назначения, а также уточненный тип сообщения о недоступности.

В качестве примера использования программы ICMP приведено несколько последних ICMP-сообщений, полученных при запуске `traceroute` (совет 35).

```
traceroute -q 1 netcom4.netcom.com
```

Опция `-q 1` означает, что `traceroute` должна посылать пробный запрос только один раз, а не три, как принято по умолчанию.

```
ICMP Истекло время существования (11) от h1-0.mig-fl-gw1.icg.net
(165.236.144.110)
   TTL == 0 во время транзита
ICMP Истекло время существования (11) от s10-0-0.dfw-tx-
gw1.icg.net (165.236.32.74)
   TTL == 0 во время транзита
ICMP Истекло время существования (11) от dfw-tx-gw2.icg.net
(163.179.1.133)
   TTL == 0 во время транзита
ICMP Пункт назначения недоступен (3) от netcom4.netcom.com
(199.183.9.104)
   Порт недоступен
   Ист. 205.184.142.71.45935, Назн. 199.183.9.104.33441
```

Обычно нет необходимости следить с помощью `icmp` за работой `traceroute`, но это может быть очень полезно для поиска причин отсутствия связи.

Резюме

В этом разделе разработан инструмент для перехвата и печати ICMP-сообщений. Такая программа помогает при диагностике ошибок сети и маршрутизации.

В ходе разработки программы `icmp` использованы простые сокеты. Здесь вы познакомились с форматами IP- и UDP-датаграмм, а также со структурой ICMP-сообщений.

Совет 41. Читайте книги Стивенса

В сетевых конференциях чаще всего задают вопрос: «Какие книги нужно читать, чтобы освоить TCP/IP?». В подавляющем большинстве ответов упоминаются книги Ричарда Стивенса.

В этой книге много ссылок на работы Стивенса. Для сетевых программистов этот автор написал две серии книг: «TCP/IP Illustrated» в трех томах и «UNIX Network Programming» в двух. Они преследуют разные цели, поэтому рассмотрим их по отдельности.

«TCP/IP Illustrated»

Как следует из названия, серия «TCP/IP Illustrated» трактует работу наиболее распространенных протоколов из семейства TCP/IP и программ, в которых они применяются. В совете 14 говорилось, что основное средство для исследования – это программа `tcpdump`. Запуская небольшие тестовые программы и наблюдая за генерируемым ими сетевым трафиком, вы постепенно начинаете понимать, как на практике функционируют протоколы.

Используя различные операционные системы, Стивенс показывает, что реализации в них одного и того же протокола приводят к тонким отличиям в его работе. Еще важнее, что вы учитесь ставить собственные эксперименты, а затем интерпретировать их результаты, отвечая тем самым на возникающие вопросы.

Поскольку в каждом томе семейство протоколов TCP/IP рассматривается под разными углами зрения, имеет смысл кратко охарактеризовать каждую книгу.

Том 1: Протоколы

В этом томе описываются классические протоколы TCP/IP и их взаимосвязи. Сначала рассматриваются протоколы канального уровня, такие как Ethernet, SLIP и PPP. Далее автор переходит к протоколам ARP и RARP (Reverse Address Resolution Protocol – протокол определения адреса по местоположению узла сети) и рассматривает их в качестве связующего звена между канальным и межсетевым уровнями.

Несколько глав посвящено протоколу IP и его связям с ICMP и маршрутизацией. Также анализируются утилиты ping и traceroute, работающие на уровне IP.

Далее речь идет о протоколе UDP и смежных вопросах: широковещании и протоколе IGMP. Описываются также основанные на UDP протоколы: DNS, TFTP (Trivial File Transfer Protocol – тривиальный протокол передачи файлов) и BOOTP (Bootstrap Protocol – протокол начальной загрузки по сети).

Восемь глав посвящено протоколу TCP. В нескольких главах обсуждаются распространенные приложения на базе TCP, такие как telnet, rlogin, FTP, SMTP (электронная почта) и NFS.

Том 2: Реализация

Второй том, написанный в соавторстве с Гэри Райтом (Gary Wright), – это практически построчное описание сетевого кода из операционной системы 4.4BSD. Поскольку код из системы BSD широко признан как эталонная реализация, эта книга незаменима для тех, кто хочет лучше разбираться в реализации основных протоколов семейства TCP/IP.

В книге рассматривается реализация нескольких протоколов канального уровня (Ethernet, SLIP и возвратный интерфейс), протокола IP, маршрутизации, протоколов ICMP, IGMP, UDP и TCP, группового вещания, уровня сокетов, а также несколько смежных тем. Поскольку автор приводит реальный код, читатель может получить представление о том, какие проблемы возникают при реализации сложной сетевой системы, и на какие компромиссы приходится идти.

Том 3: TCP для транзакций, HTTP, NNTP и протоколы в адресном домене UNIX

Третий том – это продолжение первого и второго. Он начинается с описания протокола T/TCP и принципов его функционирования. Это описание построено так же, как и в первом томе. Далее приводится реализация T/TCP – по типу второго тома.

Во второй части рассматриваются два популярных прикладных протокола: HTTP (Hypertext Transfer Protocol – протокол передачи гипертекста) и NNTP (Network News Transfer Protocol – сетевой протокол передачи новостей), которые составляют основу сети World Wide Web и сетевых телеконференций Usenet соответственно.

И, наконец, исследуются сокеты в адресном домене UNIX и их реализация. По сути, это продолжение второго тома, не включенное в него из-за ограничений на объем издания.

«UNIX Network Programming»

В серии «UNIX Network Programming» приведена трактовка TCP/IP для прикладных программистов. Здесь рассматриваются не сами протоколы, а их применение для построения сетевых приложений.

Том 1. Сетевые API: Сокеты и XTI

Эта книга должна быть у каждого сетевого программиста. В ней очень подробно рассматривается программирование TCP/IP с помощью API сокетов и XTI. Помимо традиционных тем, обсуждаемых в изданиях по программированию в архитектуре клиент-сервер, в данной книге затрагиваются групповое вещание, маршрутизирующие сокеты, неблокирующий ввод/вывод, протокол IPv6 и его работу совместно с IPv4, простые сокеты, программирование на канальном уровне и сокета в адресном домене UNIX.

В этом томе есть особенно ценная глава, в которой сравниваются различные модели построения клиентов и серверов. В приложениях описываются виртуальные сети и техника отладки.

Том 2: Межпроцессное взаимодействие

Во втором томе детально рассмотрены различные механизмы межпроцессного взаимодействия. Помимо таких традиционных средств, как каналы и FIFO в UNIX, очереди сообщений, семафоры и разделяемая память, впервые появившиеся в системе SysV, обсуждаются и более современные методы межпроцессного взаимодействия, предложенные в стандарте POSIX.

Имеется прекрасное введение в изучение стандартизованных POSIX-потоков (threads) и использования в них таких примитивов синхронизации, как мьютексы, условные переменные и блокировки чтения-записи. Для тех, кто интересуется работой системных механизмов, Стивенс приводит реализацию нескольких примитивов синхронизации и очередей сообщений в стандарте POSIX.

Заканчивается книга главами об RPC (Remote Procedure Calls – вызовы удаленных процедур) и подсистеме Solaris Doors.

Был запланирован и третий том, в котором предполагалось рассмотреть приложения, но, к несчастью, Стивенс скончался, не успев его завершить. Частично материал, который он хотел включить в третий том, можно найти в первом издании книги «UNIX Network Programming» [Stevens 1990].

Совет 42. Читайте тексты программ

Начинающие программисты часто спрашивают более опытных коллег, откуда тем столько всего известно. Конечно, знания и опыт приобретаются разными способами, но один из самых важных, хотя и недооцениваемых, – это чтение программ, написанных мастерами.

Расширяющееся движение за открытые исходные тексты облегчает эту задачу. Чтение и разбор высококачественного кода имеют множество плюсов, самое очевидное – это ознакомление с подходом, выбранным экспертом для решения задачи. Вы можете применить такую методику в своих программах, немного ее модифицировав и адаптировав. При этом вырабатываются собственные приемы. 1И когда-нибудь будут читать уже ваш код и восхищаться красивым решеним.

Не так очевидно, хотя в некоторых отношениях более важно осознание того, что нет никакой магии. Начинающие программисты иногда склонны думать, что код операционной системы или реализации протоколов непостижим, он создается высшими силами, а простым смертным нечего и пытаться в нем разобраться. Но, читая код, вы понимаете, что это просто образец хорошей (по большей части, стандартной) практики инженерного проектирования, и вам это тоже под силу.

Короче говоря, изучая код, вы приходите к выводу, что глубокое и таинственное – в действительности вопрос применения стандартных приемов, овладеваете этими приемами и учитесь применять их на практике. Читать код нелегко. Для этого требуется высокая концентрация внимания, но усилия окупаются сторицей.

Есть несколько источников хорошего кода, но лучше получить еще и комментарии. Книга Лионса «A Commentary on the UNIX Operating System» [Lions 1977] давно уже ходила в списках. Недавно благодаря усилиям нескольких людей, в частности Денниса Ричи, и великодушию компании SCO, которая сейчас владеет исходными текстами UNIX, эта книга стала доступна широкой публике.

Примечание *Первоначально книгу могли приобрести только держатели лицензии на исходные тексты UNIX, но подпольная ксерокопия (или копия с ксерокопии) была вожделенным призом для многих программистов в дни становления UNIX.*

В книге приведен код очень ранней (шестой) версии операционной системы UNIX, в которой не было сетевых компонент, кроме TTY-терминалов с разделением времени. Тем не менее стоит изучить этот код, даже если вы не интересуетесь системой UNIX, поскольку это прекрасный пример конструирования программного обеспечения

Еще одна отличная книга по операционным системам, включающая исходные тексты, – это «Operating Systems: Design and Implementation» [Tanenbaum and Woodhull, 1997]. В ней описана операционная система MINIX. Хотя в самом тексте сетевой код не приводится, но он есть на прилагаемом компакт-диске.

Для тех, кого больше интересуют сетевые задачи, предназначен второй том книги «TCP/IP Illustrated» [Wright and Stevens 1995]. Она упоминалась в совете 41.

В этой книге описывается код из системы BSD, на базе которой создано несколько современных систем с открытыми исходными текстами (FreeBSD, OpenBSD, NetBSD). Она дает прекрасный материал для экспериментов с кодом. Оригинальный код системы 4.4BSD Lite можно получить с FTP-сервера компании Walnut Creek CD-ROM (ftp://ftp.cdrom.com/pub/4.4BSD-Lite).

Во втором томе книги «Работа в сетях: TCP/IP» [Comer and Stevens 1999] описан другой стек TCP/IP. Как и в предыдущей, в ней приводится подробное объяснение принципа работы кода. Код можно загрузить из сети.

Есть много и других источников кода, хотя, как правило, он не сопровождается пояснениями в виде книги. Начать можно с открытых систем UNIX или Linux. Для всех подобных проектов исходные тексты доступны на CD-ROM или через FTP.

В проекте GNU, основанном фондом Free Software Foundation, имеется исходный текст переписанных с нуля реализаций большинства стандартных утилит UNIX. Это тоже отличный материал для изучения.

Информацию об этих проектах можно найти на следующих сайтах:

- домашняя страница FreeBSD http://www.freebsd.org;
- проект GNU http://www.gnu.org;
- архивы ядер Linux http://www.kernel.org;
- домашняя страница NetBSD http://www.netbsd.org;
- домашняя страница OpenBSD http://www.openbsd.org.

В каждом из этих источников есть огромное количество исходных текстов, связанных с сетевым программированием, и их стоит изучить, даже если UNIX не находится в сфере ваших интересов.

Резюме

Один из лучших способов изучения сетевого программирования (да и любого другого) – это чтение программ, написанных людьми, уже достигшими вершин мастерства. До недавнего времени было нелегко получить доступ к исходным текстам операционных систем и их сетевых подсистем. Но в связи с распространением движения за открытость исходных текстов ситуация изменилась. Код нескольких реализаций стека TCP/IP и соответствующих утилит (telnet, FTP, inetd и т.д.) доступен для проектов FreeBSD и Linux. Здесь приведены лишь некоторые источники, в Internet можно найти множество других.

Особенно полезны книги, в котрых есть не только код, но и подробные комментарии к нему.

Совет 43. Изучайте RFC

Ранее говорилось, что спецификации семейства протоколов TCP/IP и связанные с ними архитектурные вопросы Internet содержатся в серии документов, объединенных названием Request for Comments (RFC – Предложения для обсуждения). На самом деле, RFC, впервые появившиеся в 1969 году, – это не только спецификации протоколов. Их можно назвать рабочими документами, в которых обсуждаются разнообразные аспекты компьютерных коммуникаций и сетей. Не все RFC чисто технические, в них встречаются забавные наблюдения, пародии, стихи и просто различные высказывания. К концу 1999 года было более 2000 присвоенных RFC номеров, правда, некоторые из них так и не были опубликованы.

Хотя не в каждом RFC содержится какой-либо стандарт Internet, любой стандарт Internet опубликован в виде RFC. Материалам, входящим в подсерию RFC, дается дополнительная метка «STDxxxx». Текущий список стандартов и тех RFC, которые находятся на пути принятия в качестве стандарта, опубликован в документе STD0001.

Не следует, однако, думать, что RFC, не упомянутые в документе STD0001, лишены технической ценности. В некоторых описываются идеи пока еще разрабатываемых протоколов или направления исследовательских работ. Другие содержат информацию или отчеты о деятельности многочисленных рабочих групп, созданных по решению IETF (Internet Engineering Task Force – проблемная группа проектирования Internet).

Тексты RFC

Получить копии RFC можно разными путями, но самый простой – зайти на Web-страницу редактора RFC http://www.rfc-editor.org. На этой странице есть основанное на заполнении форм средство загрузки, значительно упрощающее поиск. Есть также поиск по ключевым словам, позволяющий найти нужные RFC, если их номер неизвестен. Там же можно получить документы из подсерий STD, FYI и BCP (Best Current Practices – лучшие современные решения).

RFC можно также переписать по FTP с сайта ftp.isi.edu из каталога in-notes/ и из других FTP-архивов.

Если у вас нет доступа по протоколам HTTP или FTP, то можно заказать копии RFC по электронной почте. Подробные инструкции о том, как сделать заказ, а также список FTP-сайтов вы получите, послав электронное сообщение по адресу rfc-info@isi.edu, включив одну строку:

```
help: ways_to_get_rfcs
```

Какой бы способ вы ни выбрали, прежде всего надо загрузить текущий указатель RFC (файл `rfc-index.txt`). После публикации ни номер, ни текст RFC уже не изменяются, так что единственный способ модифицировать RFC – это выпустить другое RFC, заменяющее предыдущее. Для каждого RFC в указателе отмечено, есть ли для него заменяющее RFC и если есть, то его номер. Там же указаны RFC, которые обновляют, но не замещают прежние.

И, наконец, различные компании поставляют RFC на CD. Так, Walnut Creek CD-ROM (http://www.cdrom.com) и InfoMagic (http://www.infomagic.com) предлагают компакт-диски, на которых записаны как RFC, так и другие документы, относящиеся к Internet. Разумеется, перечень RFC на таких дисках быстро становится неполным, но, поскольку RFC сами по себе не подлежат изменению, диск может устареть только в том смысле, что не содержит последних RFC.

Совет 44. Участвуйте в конференциях Usenet

Одно из самых ценных мест в Internet в плане получения советов и информации – это конференции Usenet, посвященные сетевому программированию.

Существуют конференции практически по любому аспекту сетевых технологий от прокладки кабелей (comp.dcom.cabling) до синхронизирующего сетевого протокола NTP (comp.protocols.time.ntp).

Замечательная конференция, относящаяся к протоколам семейства TCP/IP и программированию с их помощью, – comp.protocols.tcp-ip. Всего лишь несколько минут, ежедневно потраченных на просмотр сообщений в этой конференции, даст массу полезной информации, советов и приемов. Обсуждаемые темы варьируются от подключения к сети машины под управлением Windows до тонких технических вопросов по протоколам TCP/IP, их реализации и работы.

В самом начале знакомства с конференциями по сетям вызвать недоумение может даже простое их перечисление (а их не меньше 70). Лучше всего начать с конференции comp.protocols.tcp-ip и, возможно, одной из конференций по конкретной операционной системе, например, comp.os.linux.networking или comp.ms-windows.programmer.tools.winsock. Сообщения в этих конференциях могут содержать ссылки на другие, более специальные конференции, которые тоже могут быть вам интересны или полезны.

Один из лучших способов чему-то научиться в сетевых конференциях – это отвечать на вопросы. Изложив свои знания по конкретному вопросу в форме, понятной для других, вы сами глубже разберетесь в проблеме. Небольшое усилие, требуемое для составления ответа из 50–100 слов, окупится многократно.

Отличное введение в систему конференций Usenet находится на сайте Информационного центра Usenet (http://metalab.unc.edu/usenet-i/). На этом сайте есть статьи по истории и использованию Usenet, а также краткая статистика для большинства конференций, в том числе среднее число сообщений в день, среднее число читателей, адрес модератора (если таковой есть), где хранится архив (если он ведется) и ссылки на часто задаваемые вопросы (FAQ) для каждой конференции.

На сайте Информационного центра Usenet есть поисковая система, позволяющая найти конференции, в которых обсуждается интересующая вас тема.

Выше упоминалось, что во многих конференциях ведутся FAQ, с которыми стоит ознакомиться, прежде чем задавать вопрос. Если задать вопрос, на который уже есть ответ в FAQ, то, скорее всего, вас к нему и отошлют, а, может быть, и пожурят за нежелание потрудиться самому.

Другие ресурсы, относящиеся к конференциям

Следует упомянуть еще о двух ценных ресурсах, связанных с сетевыми конференциями. Первый – это сайт DejaNews (http://www.deja.com).

Примечание *В мае 1999 сайт Deja News изменил свое название на Deja.com. Владельцы объясняют это расширением спектра услуг. В этом разделе говорится только о первоначальных услугах по архивации сообщений из сетевых конференций и поиску в архивах.*

На этом сайте хранятся архивы примерно 45000 дискуссионных форумов, включая конференции Usenet и собственные конференции Deja Community Discussions. Владельцы Deja.com утверждают, что примерно две трети всех архивов

составляют сообщения из конференций Usenet. На конец 1999 года в архивах хранились сообщения, начиная с марта 1995 года.

Поисковая система сайта Power Search позволяет искать ответ на конкретный вопрос или информацию по некоторой проблеме в отдельной конференции, в группе или даже во всех конференциях по ключевому слову, теме, автору или диапазону дат. Второй ценный ресурс – это список ресурсов по TCP/IP (TCP/IP Resources List) Юри Раца (Uri Raz), который каждые две недели рассылается в конференцию comp.protocols.tcp-ip и некоторые более специальные. Этот список – отличная отправная точка для тех, кто ищет конкретную информацию или общий обзор TCP/IP и соответствующих API.

В списке имеются ссылки на книги по этому вопросу и другим, касающимся сетей; онлайновые ресурсы (к примеру, страницы IETF и сайты, где размещаются FAQ); онлайновые книги и журналы, учебники по TCP/IP; источники информации по протоколу IPv6; домашние страницы многих популярных книг по сетям; домашние страницы книжных издательств; домашние страницы проекта GNU и открытых операционных систем; поисковые машины с описанием способов работы с ними и конференции, посвященные сетям.

Самая последняя редакция списка находится на сайтах:

- http://www.private.org.il/tcpip_rl.html;
- http://www.best.com.il/~mphunter/tcpip_resources.html.

Также информация может быть загружена по FTP с сайтов:

- ftp://rtfm.mit.edu/pub/usenet-by-group/news.answers/internet/tcp-ip/resource-list;
- ftp://rtfm.mit.edu/pub/usenet-by-hierarchy/comp/protocols/tcp-ip/TCP-IP_Resources_List.

Особую ценность списку ресурсов по TCP/IP придает тот факт, что автор регулярно обновляет его. Это немаловажно, так как ссылки в Web имеют тенденцию быстро устаревать.

Приложение 1

Вспомогательный код для UNIX

Заголовочный файл etcp.h

Почти все программы в этой книге начинаются с заголовочного файла `etcp.h` (листинг П1.1). Он подключает и другие необходимые файлы, в том числе `skel.h` (листинг П2.1), а также определения некоторых констант, типов данных и прототипов.

Листинг П1.1. Заголовочный файл etcp.h

etcp.h

```
#ifndef __ETCP_H__
#define __ETCP_H__
/* Включаем стандартные заголовки. */
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <stdarg.h>
#include <string.h>
#include <netdb.h>
#include <signal.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/stat.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "skel.h"
#define TRUE    1
#define FALSE   0
#define NLISTEN 5   /* Максимальное число ожидающих соединений. */
#define NSMB    5   /* Число буферов в разделяемой памяти. */
#define SMBUFSZ 256 /* Размер буфера в разделяемой памяти. */
extern char *program_name;     /* Для сообщений об ошибках. */
#ifdef __SVR4
```

```
#define bzero(b,n) memset( ( b ), 0, ( n ) )
#endif

typedef void ( *tofunc_t )( void * );

void error( int, int, char*, ... );
int readn( SOCKET, char *, size_t );
int readvrec( SOCKET, char *, size_t );
int readcrlf( SOCKET, char *, size_t );
int readline( SOCKET, char *, size_t );
int tcp_server( char *, char * );
int tcp_client( char *, char * );
int udp_server( char *, char * );
int udp_client( char *, char *, struct sockaddr_in * );
int tselect( int, fd_set *, fd_set *, fd_set *);
unsigned int timeout( tofunc_t, void *, int );
void untimeout( unsigned int );
void init_smb( int );
void *smballoc( void );
void smbfree( void * );
void smbsend( SOCKET, void * );
void *smbrecv( SOCKET );
void set_address( char *, char *, struct sockaddr_in *, char * );
#endif  /* __ETCP_H__ */
```

etcp.h

Функция daemon

Функция daemon, которая использована в программе tcpmux, входит в стандартную библиотеку, поставляемую с системой BSD. Для систем SVR4 приводится версия, текст которой показан в листинге П1.2.

Листинг П1.2. Функция daemon

daemon.c

```
int daemon( int nocd, int noclose )
{
    struct rlimit rlim;
    pid_t pid;
    int i;

    umask( 0 );      /* Очистить маску создания файлов. */

    /* Получить максимальное число открытых файлов. */

    if ( getrlimit( RLIMIT_NOFILE, &rlim ) < 0 )
        error( 1, errno, "getrlimit failed" );

    /* Стать лидером сессии, потеряв при этом управляющий терминал... */

    pid = fork();
    if ( pid < 0 )
        return -1;
    if ( pid != 0 )
        exit( 0 );
```

```
  setsid();
  /* ... и гарантировать, что больше его не будет. */
  signal( SIGHUP, SIG_IGN );
  pid = fork();
  if ( pid < 0 )
    return -1;
  if ( pid != 0 )
    exit( 0 );
  /* Сделать текущим корневой каталог, если не требовалось обратное */
  if ( !nocd )
    chdir( "/" );
  /*
   * Если нас не просили этого не делать, закрыть все файлы.
   * Затем перенаправить stdin, stdout и stderr
   * на /dev/null.
   */
  if ( !noclose )
  {
#if 0 /* Заменить на 1 для закрытия всех файлов. */
  if ( rlim.rlim_max == RLIM_INFINITY )
    rlim.rlim_max = 1024;
  for ( i = 0; i < rlim.rlim_max; i++ )
    close( i );
#endif
  i = open( "/dev/null", O_RDWR );
  if ( i < 0 )
    return -1;
  dup2( i, 0 );
  dup2( i, 1 );
  dup2( i, 2 );
  if ( i > 2 )
    close( i );
}
return 0;
}
```

daemon.c

Функция signal

В этой книге уже упоминалось, что в некоторых версиях UNIX функция `signal` реализована на основе семантики ненадежных сигналов. В таком случае для получения семантики надежных сигналов следует использовать функцию `sigaction`. Чтобы повысить переносимость, необходимо реализовать `signal` с помощью `sigaction` (листинг П1.3)

Листинг П1.3. Функция signal

signal.c

```
/* signal - надежная версия для SVR4 и некоторых других систем. */
typedef void sighndlr_t( int );
sighndlr_t *signal( int sig, sighndlr_t *hndlr )
{
   struct sigaction act;
   struct sigaction xact;

   act.sa_handler = hndlr;
   act.sa_flags = 0;
   sigemptyset( &act.sa_mask );
   if ( sigaction( sig, &act, &xact ) < 0 )
      return SIG_ERR;
   return xact.sa_handler;
}
```

signal.c

Приложение 2

Вспомогательный код для Windows

Заголовочный файл skel.h

Для компиляции примеров программ на платформе Windows вы можете пользоваться тем же файлом etcp.h, что и для UNIX (листинг П1.1). Вся системно зависимая информация находится в заголовочном файле skel.h, версия которого для Windows приведена в листинге П2.1.

Листинг П2.1. Версия skel.h для Windows

skel.h

```
#ifndef __SKEL_H__
#define __SKEL_H__

/* Версия Winsock. */

#include <windows.h>
#include <winsock2.h>

struct timezone
{
   long tz_minuteswest;
   long tz_dsttime;
};
typedef unsigned int u_int32_t;

#define EMSGSIZE           WSAEMSGSIZE
#define INIT()             init( argv );
#define EXIT(s)            do { WSACleanup(); exit( ( s ) ); } \
                           while ( 0 )
#define CLOSE(s)           if ( closesocket( s ) ) \
                           error( 1, errno, "ошибка вызова close")
#define errno              ( GetLastError() )
#define set_errno(e)       SetLastError( ( e ) )
#define isvalidsock(s)     ( ( s ) != SOCKET_ERROR )
#define bzero(b,n)  memset ( ( b ), 0, ( n ) )
#define sleep(t)           Sleep( ( t ) * 1000 )
#define WINDOWS

#endif  /* __SKEL_H__ */
```

skel.h

Функции совместимости с Windows

В листинге П2.2 приведены различные функции, которые использованы в примерах, но отсутствуют в Windows.

Листинг П2.2. Функции совместимости с Windows

wincompat.c

```
#include <sys/timeb.h>
#include "etcp.h"
#include <winsock2.h>

#define MINBSDSOCKERR           ( WSAEWOULDBLOCK )
#define MAXBSDSOCKERR           ( MINBSDSOCKERR + \
                                ( sizeof( bsdsocketerrs ) / \
                                sizeof( bsdsocketerrs[ 0 ] ) ) )

extern int sys_nerr;
extern char *sys_errlist[];
extern char *program_name;
static char *bsdsocketerrs[] =
{
"Resource temporarily unavailable",    /* Ресурс временно недоступен. */
"Operation now in progress",           /* Операция начала выполняться. */
"Operation already in progress",       /* Операция уже выполняется. */
"Socket operation on non-socket",      /* Операция сокета не над сокетом. */
"Destination address required",        /* Нужен адрес назначения. */
"Message too long",                    /* Слишком длинное сообщение. */
"Protocol wrong type for socket",      /* Неверный тип протокола для сокета. */
"Bad protocol option",                 /* Некорректная опция протокола. */
"Protocol not supported",              /* Протокол не поддерживается. */
"Socket type not supported",           /* Тип сокета не поддерживается. */
"Operation not supported",             /* Операция не поддерживается. */
"Protocol family not supported",       /* Семейство протоколов не */
                                       /* поддерживается. */
"Address family not supported by protocol family", /* Адресное семейство */
                           /* не поддерживается семейством протоколов*/
"Address already in use",              /* Адрес уже используется. */
"Can't assign requested address",      /* Не могу выделить затребованный */
                                       /* адрес. */
"Network is down",                     /* Сеть не работает. */
"Network is unreachable",              /* Сеть недоступна. */
"Network dropped connection on reset", /* Сеть сбросила соединение */
                                       /* при перезагрузке. */
"Software caused connection abort",    /* Программный разрыв соединения. */
"Connection reset by peer",            /* Соединение сброшено другой */
                                       /* стороной. */
"No buffer space available",           /* Нет буферов. */
"Socket is already connected",         /* Сокет уже соединен. */
"Socket is not connected",             /* Сокет не соединен. */
"Cannot send after socket shutdown",   /* Не могу послать данные после */
                                       /* размыкания. */
"Too many references: can't splice",  /* Слишком много ссылок. */
"Connection timed out",                /* Таймаут на соединении. */
"Connection refused",                  /* В соединении отказано. */
"Too many levels of symbolic links",   /* Слишком много уровней */
                                       /* символических ссылок. */
"File name too long",                  /* Слишком длинное имя файла. */
"Host is down",                        /* Хост не работает. */
"No route to host"                     /* Нет маршрута к хосту. */
};
void init( char **argv )
```

```
{
    WSADATA wsadata;
    ( program_name = strrchr( argv[ 0 ], '\\' ) ) ?
        program_name++ : ( program_name = argv[ 0 ] );
    WSAStartup( MAKEWORD( 2, 2 ), &wsadata );
}
/* inet_aton - версия inet_aton для SVr4 и Windows. */
int inet_aton( char *cp, struct in_addr *pin )
{
   int rc;
    rc = inet_addr( cp );
    if ( rc == -1 && strcmp( cp, "255.255.255.255" ) )
        return 0;
    pin->s_addr = rc;
    return 1;
}
/* gettimeofday - для tselect. */
int gettimeofday( struct timeval *tvp, struct timezone *tzp )
{
    struct _timeb tb;
    _ftime( &tb );
    if ( tvp )
    {
        tvp->tv_sec = tb.time;
        tvp->tv_usec = tb.millitm * 1000;
    }
    if ( tzp )
    {
        tzp->tz_minuteswest = tb.timezone;
        tzp->tz_dsttime = tb.dstflag;
    }
}
/* strerror - версия, включающая коды ошибок Winsock. */
char *strerror( int err )
{
    if ( err >= 0 && err < sys_nerr )
        return sys_errlist[ err ];
    else if ( err >= MINBSDSOCKERR && err < MAXBSDSOCKERR )
        return bsdsocketerrs[ err - MINBSDSOCKERR ];
    else if ( err == WSASYSNOTREADY )
        return "Network subsystem is unusable";
                    /* СОетевая подсистема неработоспособна. */
    else if ( err == WSAVERNOTSUPPORTED )
        return "This version of Winsock not supported";
                    /* Эта версия Winsock не поддерживается. */
    else if ( err == WSANOTINITIALISED )
        return "Winsock not initialized";
                    /* Winsock не инициализирована. */
    else
        return "Unknown error";
                    /* Неизвестная ошибка. */
}
```

wincompat.c

Литература

Albitz, P. and Liu, C. 1998. *DNS and BIND, 3rd Edition.* O'Reilly & Associates, Sebastopol, Calif.

Baker, R, ed. 1995. «Requirements for IP Version 4 Routers», RFC 1812 (June).

Banga, G. and Mogul, J. C. 1998. «Scalable Kernel Performance for Internet Servers Under Realistic Loads», *Proceedings of the 1998 USENIX Annual Technical Conference,* New Orleans, LA.

http://www.cs.rice.edu/~gaurav/my_papers/usenix98.ps

Bennett, J. C. R., Partridge, C., and Shectman, N. 1999. «Packet Reordering Is Not Pathological Network Behavior», *IEEE/ACM Transactions on Networking,* vol. 7, no. 6, pp. 789-798 (Dec.).

Braden, R. T. 1985. «Towards a Transport Service for Transaction Processing Applications», RFC 955 (Sept.).

Braden, R. T. 1992a. «Extending TCP for Transactions—Concepts,» RFC 1379 (Nov.).

Braden, R. T. 1992b. «TIME-WAIT Assassination Hazards in TCP,» RFC 1337 (May).

Braden, R. T. 1994. «T/TCP—TCP Extensions for Transactions, Functional Specification», RFC 1644 (July).

Braden, R. T., ed. 1989. «Requirements for Internet Hosts—Communication Layers», RFC 1122 (Oct.).

Brown, C. 1994. *UNIX Distributed Programming.* Prentice Hall, Englewood Cliffs, N.J.

Castro, E. 1998. *Peri and CGI for the World Wide Web: Visual QuickStart Guide.* Peachpit Press, Berkeley, Calif.

Clark, D. D. 1982. «Window and Acknowledgement Strategy in TCP», RFC 813 (July).

Cohen, D. 1981. «On Holy Wars and a Plea for Peace», *IEEE Computer Magazine,* vol. 14, pp. 48-54 (Oct.).

Comer, D. E. 1995. *Internetworking with TCP/IP Volume I: Principles, Protocols, and Architecture, Third Edition.* Prentice Hall, Englewood Cliffs, N.J.

Comer, D. E. and Lin, J. C. 1995. «TCP Buffering and Performance Over an ATM Network» *Journal of Internetworking: Research and Experience,* vol. 6, no. 1, pp. 1–13 (Mar.).

ftp://gwen.cs.purdue.edu/pub/lin/TCP.atm.ps.Z

Comer, D. E. and Stevens, D. L. 1999. *internetworking with TCP/IP Volume II: Design, Implementation, and Internals, Third Edition.* Prentice Hall, Englewood Cliffs, N.J.

Fuller, V., Li, T., Yu, J., and Varadhan, K. 1993. «Classless Inter-Domain Routing (CIDR): An Address Assignment», RFC 1519 (Sept.).

Gallatin, A., Chase, J., and Yocum, K. 1999. «Trapeze/IP: TCP/IP at Near-Gigabit Speeds», *1999 Usenix Technical Conference (Freenix track)*, Monterey, Calif.

http://www.cs.duke.edu/ari/publications/tcpgig.ps

Haverlock, P. 2000. Private communication.

Hinden, R. M. 1993. «Applicability Statement for the Implementation of Classless Inter-Domain Routing (CIDR)», RFC 1517 (Sept.).

Huitema, C. 1995. *Routing in the Internet.* Prentice Hall, Englewood Cliffs, N.J.

International Standards Organization 1984. «OSI—Basic Reference Model», ISO 7498, International Standards Organization, Geneva.

Jacobson, V. 1988. «Congestion Avoidance and Control», *Proc. of SIGCOMM '88,* vol. 18, no. 4, pp. 314–329 (Aug.).

http://www-nrg.ee.Ibl.gov/nrg.html

Jacobson, V. 1999. «Re: Traceroute History: Why UDP?», Message-ID <79m7m4rehl ©dog.ee.lbl.gov>, Usenet, comp.protocols.tcp-ip (Feb.).

http://www.kohala.com/start/vanj99Feb08.txt

Jacobson, V., Braden, R. T., and Borman, D. 1992. «TCP Extensions for High Performance», RFC 1323 (May).

Jain, B. N. and Agrawala, A. K. 1993. *Open Systems Interconnection: Its Architecture and Protocols, Revised Edition.* McGraw-Hill, N.Y.

Kacker, M. 1998. Private communication.

Kacker, M. 1999. Private communication.

Kantor, B. and Lapsley, P. 1986. «Network News Transfer Protocol», RFC 977 (Feb.).

Kernighan, B. W. and Pike, R. 1999. *The Practice of Programming.* Addison-Wesley, Reading, Mass.

Kernighan, B. W. and Ritchie, D. M. 1988. *The C Programming Language, Second Edition.* Prentice Hall, Englewood Cliffs, N.J.

Knuth, D. E. 1998. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms, Third Edition.* Addison-Wesley, Reading, Mass.

Lehey, G. 1996. *The Complete FreeBSD.* Walnut Creek CDROM, Walnut Creek, Calif.

Lions, J. 1977. *Lions' Commentary on UNIX 6th Edition with Source Code.* Peer-to-Peer Communications, San Jose, Calif.

Lotter, M. K. 1988. «TCP Port Service Multiplexer (TCPMUX)», RFC 1078 (Nov.).

Mahdavi, J. 1997. «Enabling High Performance Data Transfers on Hosts: (Notes for Users and System Administrators)», Technical Note (Dec.).

http://www.psc.edu/networking/perf_tune.html

Malkin, G. 1993. «Traceroute Using an IP Option», RFC 1393 (Jan.).

McCanne, S. and Jacobson, V. 1993. «The BSD Packet Filter: A New Architecture for User-Level Packet Capture», *Proceedings of the 1993 Winter USENIX Conference,* pp. 259–269, San Diego, Calif.

ftp://ftp.ee.lbl.gov/papers/bpf-usenix93-ps.Z

Miller, B. P., Koski, D., Lee, C. P., Maganty, V., Murthy, R., Natarajan, A., and Steidi, J. 1995. «Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services», CS-TR-95-1268, University of Wisconsin (Apr.).

ftp://grilled.cs.wise.edu/technical_papers/fuzz-revisited.ps.Z

Minshall, G., Saito, Y., Mogul, J. C., and Verghese, B. 1999. «Application Performance Pitfalls and TCP's Nagle Algorithm», ACM SIGMETRICS Workshop on Internet Server Performance, Atlanta, Ga..

http://www.cc.gatech.edu/fac/Ellen.Zegura/wisp99/papers/minshall.ps

Mogul, J. and Postel, J. B. 1985. «Internet Standard Subnetting Procedure», RFC 950 (Aug.).

Nagle, J. 1984. «Congestion Control in IP/TCP Internetworks», RFC 896 (Jan.).

Oliver, M. 2000. Private communication.

Padlipsky, M. A. 1982. «A Perspective on the ARPANET Reference Model,» RFC 871 (Sept.).

Partridge, C. 1993. «Jacobson on TCP in 30 Instruction», Message-ID <1993Sep8.213239.28992 @sics.se>, Usenet, comp.protocols.tcp-ip Newsgroup (Sept.).

http: //www-nrg.ee. Ibl .gov/nrg-email .html

Partridge, C. and Pink, S. 1993. «A Faster UDP», *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, pp. 427–440 (Aug.).

http:/www.ir.bbn.com/~craig/udp.ps

Patchett, C. and Wright, M. 1998. *The CGI/Perl Cookbook.* John Wiley & Sons, N.Y.

Paxson, V. 1995. «Re: Traceroute and TTL», Message-ID <48407@dog.ee.lbl.gov>, Usenet, comp.protocols.tcp-ip (Sept.).

ftp://ftp.ee.Ibl.gov/email/paxson.95sep29.txt

Paxson, V. 1997. «End-to-End Routing Behavior in the Internet», *IEEE/ACM Transactions on Networking*, vol. 5, no. 5, pp. 601–615 (Oct.).

ftp://ftp.ee.Ibl.gov/papers/vp-routing-TON.ps.Z

Plummer, W. W. 1978. «TCP Checksum Function Design», IEN 45 (June). Reprinted as an appendix to RFC 1071.

Postel, J. B. 1981. «Internet Control Message Protocol», RFC 792 (Sept.).

Postel, J. B., ed. 1981a. «Internet Protocol», RFC 791 (Sept.).

Postel, J. B., ed. 1981b. «Transmission Control Protocol», RFC 793 (Sept.).

Quinn, B. and Shute, D. 1996. Windows Sockets Network Programming. Addison-Wesley, Reading, Mass.

Rago, S. A. 1993. *UNIX System V Network Programming.* Addison-Wesley, Reading, Mass.

Rago, S. A. 1996. «Re: Sockets vs TLL» Message-ID <50pcds$jl8@prologic.plc.com>, Usenet, comp.protocols.tcp-ip (Oct.).

Rekhter, Y. and Li, T. 1993. «An Architecture for IP Address Allocation with CIDR», RFC 1518 (Sept.).

Rekhter, Y, Moskowitz, R. G., Karrenberg, D., Groot, G. J. de, and Lear, E. 1996. «Address Allocation of Private Internets», RFC 1918 (Feb.).

Reynolds, J. K. and Postel, J. B. 1985. «File Transfer Protocol (FTP)», RFC 959 (Oct.).

Richter, J. 1997. *Advanced Windows, Third Edition.* Microsoft Press, Redmond, Wash.

Ritchie, D. M. 1984. «A Stream Input-Output System», *AT&T Bell Laboratories Technical Journal*, vol. 63, no. No. 8 Part 2, pp. 1897–1910 (Oct.).

http://cm.bell-labs.com/cm/cs/who/dmr/st.ps

Romkey, J. L. 1988. «A Nonstandard for Transmission of IP Datagrams Over Serial Lines: SLIP», RFC 1055 (June).

Saltzer, J. H., Reed, D. P., and Clark, D. D. 1984. «End-to-End Arguments in System Design», *ACM Transactions in Computer Science*, vol. 2, no. 4, pp. 277–288 (Nov.).

http://web.mit.edu/Saltzer/www/publications

Sedgewick, R. 1998. *Algorithms in C, Third Edition, Parts 1–4*. Addison-Wesley, Reading, Mass.

Semke, J., Mahdavi, J., and Mathis, M. 1998. «Automatic TCP Buffer Tuning», *Computer Communications Review*, vol. 28, no. 4, pp. 315–323 (Oct.).

http: //www.psc . edu /networking /ftp/papers/autotune-sigcomm9 8 .ps

Srinivasan, R. 1995. «XDR: External Data Representation Standard», RFC 1832 (Aug.).

Stevens,W. R. 1990. *UNIX Network Programming*. Prentice Hall, Englewood Cliffs.

Stevens, W. R. 1994. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, Reading, Mass.

Stevens, W. R. 1996. *TCP/IP Illustrated, Volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols*. Addison-Wesley, Reading, Mass.

Stevens, W. R. 1998. *UNIX Network Programming, Volume 1, Second Edition, Networking APIs: Sockets and XTI*. Prentice Hall, Upper Saddle River, N.J.

Stevens, W. R. 1999. *UNIX Network Programming, Volume 2, Second Edition, Interprocess Communications*. Prentice Hall, Upper Saddle River, N.J.

Stone, J., Greenwald, M., Partridge, C., and Hughes, J. 1998. «Performance of Checksums and CRC's Over Real Data», *IEEE/ACM Transactions on Networking*, vol. 6, no. 5, pp. 529-543 (Oct.).

Tanenbaum, A. S. 1996. *Computer Networks, Third Edition*. Prentice Hall, Englewood Cliffs, N.J.

Tanenbaum, A. S. and Woodhull, A. S. 1997. *Operating Systems: Design and Implementation, Second Edition*. Prentice Hall, Upper Saddle River, N.J.

Torek, C. 1994. «Re: Delay in Re-Using TCP/IP Port», Message-ID <199501010028.QAA16863 ©elf.bsdi.com>, Usenet, comp.unix.wizards (Dec.).

http://www.kohala.corn/start/torek.94dec31.txt

Unix International 1991. «Data Link Provider Interface Specification,» Revision 2.0.0, Unix International, Parsippany, N.J. (Aug.).

http://www.whitefang.com/rin/docs/dlpi-ps

http://www.opengroup.org/publications/catalog/c 811.htm

Varghese, G. and Lauck, A. 1997. «Hashed and Hierarchical Timing Wheels: Efficient Data Structures for Implementing a Timer Facility», *IEEE/ACM Transactions on Networking*, vol.5, no. 6, pp. 824–834 (Dec.).

http://www.cere-wusti.edu/~varghese/PAPERS/twheel.ps.Z

Wall, L., Christiansen, T., and Schwartz, R. L. 1996. *Programming Peri, Second Edition*. O'Reilly & Associates, Sebastopol, Calif.

WinSock Group 1997. «Windows Sockets 2 Application Programming Interface», Revision 2.2.1, The Winsock Group (May).

http: //www. stardust.com/wsresource/winsock2/ws2docs .html

Wright, G. R. and Stevens, W. R. 1995. *TCP/IP Illustrated, Volume 2: The Implementation*. Addison-Wesley, Reading, Mass.

Предметный указатель

М

О

Р

С

Т

У

Ф

L

M

N

O

P

R

S

T

U

X